Imperial College London

Imperial College London

Department of Computing

JMC MEng Individual Project

Verification of feed-forward ReLU neural networks

Author:
Lalit Maganti

Supervisor:
Alessio Lomiscuo

Second Marker:
Phillipa Gardner

Abstract

Neural networks are an area of computer science where there is a tremendous amount of research being carried out. Most research till date has focused on the training of these networks or alternatively their performance when solving problems. Research into formal verification techniques has been conspicuously absent even as its importance has grown. Only recently has serious research started but the field as a whole is still in its infancy.

This project and report investigate a subset of these neural networks, so called ReLU feed-forward neural networks, to attempt to find algorithms to verify interesting properties on them. We present a sound and complete technique to verify reachability properties on these networks using linear programming. We then develop an abstraction known as a network system which opens the gateway to temporal verification. This abstraction forms a fundamental part of our algorithm to verify temporal properties on neural networks using LTL. As well developing the theory for these ideas, we also implement these algorithms in Python and evaluate these on a wide range of problems.

Acknowledgements

As with any endeavour which consumes 6 months of a one's life, I could not have done this project alone. For this reason, I would like to thank a few people:

- My second marker, Phillipa, for her excellent advice in the direction the project should take. As you will see, I ended up not trying to tackle recurrent neural networks!

- My JMC friends; there's only 9 of us left now but we've all survived! I want to especially thank Dan, Michael and Flaviu. We went through so many group projects, deadlines, exams and reports together and I'm happy to see us all out on the other side!

- My Model UN friends; there are far too many of you (shoutout to all those in Tabasco!) to list but MUN has been a large part of my life throughout my degree and especially so this year. Special thanks for Michelle who has been especially supportive during the report writing phase and for Dijana, Xuan, Kevin, Chris and Pedra for being a constant presence in my life this year and just being great friends.

- Kevin for being a friend to me from the the very beginning till the very end. We've been through a lot together and you've played a big part in shaping me into the person I am today. I'm privileged to have you as a friend.

- My supervisor, Alessio, who has been extremely helpful throughout the project. From providing assistance whenever I asked to co-authoring the papers we wrote, I was constantly amazed at your helpfulness and intuitive grasp of anything I presented to you, even in those topics you were not familiar with. I owe a large part any success of this project to you.

- My parents. I am who I am because of you. Thank you.

Contents

1 Introduction

Computer Science as a formal academic discipline is one which has now existed for more than 60 years. The progress made during this time has been immense and the field has been revolutionised many times over. However, one area which has largely remained is the way in which computers are programmed. Notably, the Pascal and C programming languages both of which were invented in the early 1970s are in widespread use today. However, the subdiscipline known as *Machine Learning* aims to change this. Instead of explicitly setting out how a problem is to be solved, Machine Learning investigates techniques which can allow computers to *learn* solutions.

Neural networks are one such approach. Modelled after biological brains, they involve linking layers of *neurons* together. Their key advantage is that, beyond dictating the structure of the network, it is not necessary to specify the exact function that is to be computed. Instead, neural networks are *trained* using large quantities of data which allows them to learn patterns in this data and thus the definition of the function.

Traditionally, neural networks have been used to solve problems such as image classification and speech recognition with reasonably competitive (but below state of the art) levels of performance. However, in the past ten years, with the growth of computing power and data available to train these networks, their popularity has truly exploded and their potential is being fulfilled. For example, convolutional neural networks have become the de-facto technique for image recognition and LSTM networks are used by companies like Facebook and Google to carry out speech recognition in their consumer products.

Nonetheless, there are many areas of computer science where neural networks have yet to begin being used; one such area is in safety critical applications. When it comes to such applications (i.e. applications where cost of failure is very high), guarantees about the output are required. With neural networks, these guarantees have proved difficult to obtain.

The field known as *systems verification* has long existed to devise techniques to provide these guarantees for traditional computer programming. Verification involves formally

checking whether a system satisfies a certain specification. This may involve the technique of *model checking* where the system is modelled using logic and the validity of a formula which represents the *safety property* we are attempting to verify is checked. Alternatively, we may directly attempt to check whether the specification is satisfied by converting the program to a set of logical statements and then check the validity of the safety property against these statements using a theorem prover: this is the focus of *static program verification.*

Verification offers significant benefits which testing could not hope to emulate much as in traditional programming. However, devising suitable tests for neural networks is often very difficult so the gap between testing and verification is much smaller than it is in these traditional cases. This increases the the attraction of verification significantly.

With the successes of systems verification in the past decade, it is perhaps rather surprising that very little research has carried out in the topic of verification of neural networks. However, We strongly believe that the field of neural networks will soon be revolutionised with large increase in interest in verification.

Historically (and even today), networks were checked for correctness using statistical techniques rather than formal methods. However, with the explosion in uses and complexity of these networks, simply testing networks will not be enough. Signs of this shift in paradigm have already started to appear; over the past three months, several papers have appeared each offering different techniques verifying reachability properties on neural networks.

This field is one far too large to be investigated by one project alone: it would take months of research to even understand the possible directions of future work. We will simply to take the explore a small portion of this field, choosing which aspects we wish to consider while providing direction and advice for future research in this area.

1.1 Primary Objectives

The high level goal of this project is to devise techniques to verify safety properties on feed-forward neural networks. Prior to this project, many safety properties have not been formally defined, still less algorithms to carry out verification of them. Therefore, we will consider both of these aspects when defining our aims for the project.

Our objectives are as follows

1. **Verify reachability properties for feed-forward neural networks** Reachability is an important property for many applications and it is easy to see how it is also important to the field of neural networks. As reachability has not been defined to date, we will need to do this before presenting an algorithm which can perform this verification.

2. **Define a abstraction for agent-environment systems** Control problems are an interesting area where neural networks are increasingly common. These are problems where there is an agent network which responds to an environment with a feed-back loop between these two systems. To perform LTL verification, we will need to define some abstraction over this structure such that a Kripke frame and model can be defined.

3. **Verify properties by checking LTL formulae on abstraction** The ability to verify LTL formulae on the agent-environment abstraction would be a significant advancement of the state of the art as it would introduce a temporal component which is absent in reachability verification. I we were to achieve this, we would be able to verify quite complex properties which have traditionally been beyond the reach of statistical tests on the performance of neural networks.

1.2 Challenges

This project was an extremely challenging undertaking involving a high degree of complexity and a large risk of failure. The main factor causing this is the relative lack of research which has been carried out in this domain and this is reflected in the issues we came across.

Some of the biggest challenges we faced were as follows

1. **Lack of techniques against which to compare reachability** A big problem with analysing the performance of the reachability algorithm was that we did not know what constituted a good level of performance. By the time we finished work on that part of the project, only toy networks had been analysed for reachability by other papers.

   Not until much later on did two papers come out which presented more complex reachability techniques. Even then, one did not release their networks so we could not compare our work against theirs. The other paper came out too late for a comparison to be possible.

To mitigate this problem, we attempted to find a wide range of problems on which we could verify reachability and hoped that one of these would have shown performance problems. However, all of them showed good performance so we were unable to determine if any performance issues existed.

2. **Conceptual difficulties with defining agent-environment abstractions** While the final concept of network systems is reasonably elegant, drawing inspiration from the field of interpreted systems, the process to get there involved several false starts.

   Initially, our plan was to define a method to merge neural networks together but this proved to be impossible to reconcile with the way we planned to implement the LTL verification algorithm. Next we had two networks; one for agent and one for the environment. However, we == found that this made the resulting structure much to complex to understand.

   Finally, we abandoned the concept of representing the environment using a single network and instead required that it be a linear combination of non-linear functions. We believe this is a good compromise between the elegance of network systems while also keeping the environment function quite general.

3. **Complexity of state approximation techniques** The method for approximating state sets we present was very complex to invent. Even in two dimensions, one can see that the details of the algorithm are quite subtle. In n-dimensions, this becomes even harder to reason about and required a lot of thought to be certain that the algorithm could be generalised.

   While we do not actually present the n-dimensional method of finding the additional approximation constraints, we believe we have given a good intuition of the method to generate them. Further research should allow for this to be formalised.

4. **Lack of techniques to compare and verify correctness of LTL verification** Similar to our problems of lack of techniques to compare to for reachability verification, we found similar problems with our LTL verification algorithm. In some ways, our problem here was worse as the correctness of the algorithm was also under investigation. The lack of a technique to compare with significantly limited our confidence in this regard as well as leaving us unable to understand in absolute terms how efficient our algorithm was.

   We have tried to mitigate this by trying to explicitly compute successor sets and compare our approximations but we believe a more rigorous investigation is re-

quired to be fully coonfident.

1.3 Achievements

The main contributions from our project are as follows

1. **Verifying reachability on feed-forward neural networks** We define the very concept of reachability on neural networks. We encode ReLU feed-forward neural networks using linear constraints. We prove that our encoding was equivalent to the function computed by the network. We define a linear program encoding reachability for a ReLU feed-forward network. We prove an equivalence between verifying reachability and solving the linear program. We give an alternative linear encoding of networks using SOS constraints. We give an encoding which took into account floating point limitations. We finish with a pseudo-code algorithm which performed reachability verification.

2. **Implementation of reachability verification** We implement the pseudo-code algorithm we presented in Python. We utilise Gurobi to solve the generated LPs. We present several case studies, both control problems which have been the subject of intense research in recent years but also classification problems where neural networks have traditionally been used. We also give benchmarks and demonstrated the efficiency of the algorithm.

3. **Network systems** We present the novel technique of modelling control problems using agent networks and environment functions. We define how these networks can be composed into a structure called a network system.

4. **Approximation-based LTL verification on network systems** We present a way to approximate the output of a network system using linear constraints. We present a linear program which can be solved to generate additional constraints in 2-D. We present an algorithm which generates the approximation and was fully defined in 2-D. We finish with an sound algorithm which can perform LTL verification using these approximated sets.

5. **Proof of concept of verifying LTL formulae using approximations** While not implementing the full LTL algorithm, we implement significant portions of it including the full approximation code for 2-D. We test this code by presenting a case study for the inverted pendulum control problem.

We attempt to demonstrate correctness of our implementation as well as giving a collection of examples where our algorithm was successful in verifying the property of stability. We also demonstrate that the additional approximation constraints are an important addition to have useful verification.

6. **Implemented a library for ReLU FFNN verification named ReVerify** We consolidate some of the useful routines from the code above and produce a library called ReVerify. Specifically, this library contains routines for generating constraints for FFNNs which is used in both reachability verification but also verification of LTL. It also contains an implementation for 2-D state sets used by our LTL work.

2 Background

In this section, we consider the state of the art from the fields the the project is based upon. Due to the broad range of areas we will discuss, we will be succinct with commonly known properties instead focusing our attention on details which are lesser known or vital to the project.

We begin with the field of neural networks which can be split into two categories based on the *topology* of the network: feed-forward and recurrent. For the purposes of this project, we focus only on feed-forward networks.

Next, we give an overview of the field of Linear Programming which forms the cornerstone of all techniques discussed in the project. We focus on the important aspects of the Gurobi solver which was used in some parts of the project.

Our last topic is systems verification: after introducing the field, we consider the popular temporal specification language LTL which is used to encode specifications to check for network systems. We then conclude by examining the work done to date to bring together these disparate domains together.

2.1 Neural networks

Informally, neural networks are computational systems modelled on human brains [26]. This style of computation is very different to the sequential execution of CPU instructions which traditional programs require.

Instead, neural networks are composed of small computational units known as *neurons* which compute a linear transformation of their input (usually a vector of real numbers but may also be a vector of binary values) followed by applying a (non-linear) *activation function*. One way of representing a neuron diagrammatically is in the form of a block diagram: for a arbitrary neuron this is given by Figure 2.1. Each neuron has a *weight* for each of its inputs which are used to compute the linear transformation. Often a *bias* term is also used to change the linear transformation into an *affine* one.

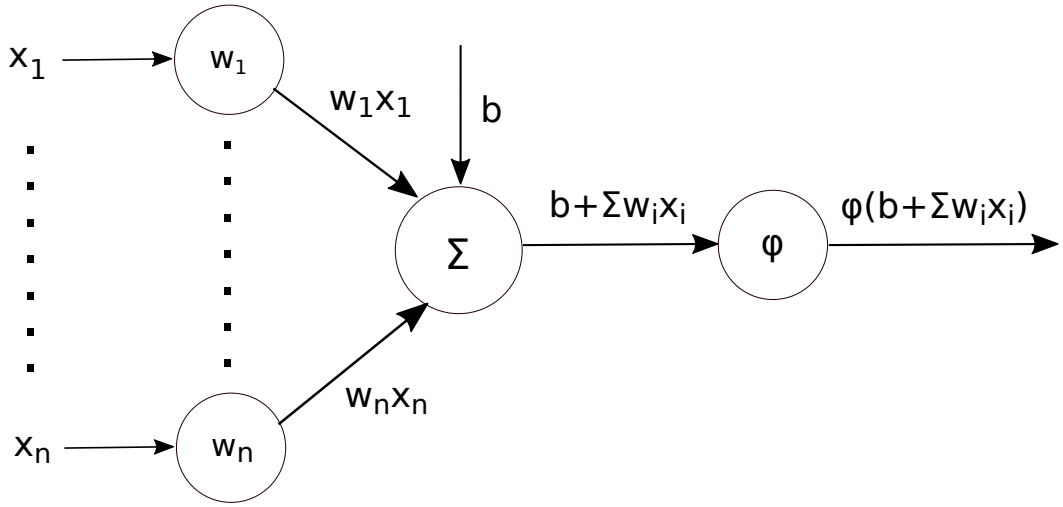


Figure 2.1: Block diagram of a single neuron in a neural network. x_i are the inputs the the neurons and are usually real numbers. w_i are the weights whose values are learned during training. b is a bias which is also learned during training. ϕ is the activation function - usually it is a non linear function.

However, when large numbers neurons are linked together to form a network, treating each neuron as an individual component can become quite unwieldy. Moreover, associating the weights with the neuron rather than with the links between neurons means that we cannot perform efficient matrix-vector multiplications when calculating the output of the network.

An equivalent view of a neural network is as a graph with the neurons acting as the vertices of the graph and weights as the edges. This view is the one we use when we define feed-forward neural networks.

The overarching purpose of a neural network is to approximate functions. In fact, neural networks with just a single hidden layer are *universal function approximators*; that is, such a neural network can approximate an arbitrary function to arbitrary accuracy.

In general, a neural network will try to approximate a function a function $f : I \to \mathbb{R}^n$ usually with $I = \mathbb{R}^m$. However, while the exact function itself is unknown, for some $J \subseteq I$, the set $\{f(j) \mid j \in J\}$ is known. Neural networks take this known data as *training data* and use this to *learn* the true function. Learning occurs by modifying the values of the weights of the network; these weights dictate the function is being represented by the network.

Neural networks can largely be broken up into three categories based on the structure of the network [26]:

1. Single-layer feed-forward networks
2. Multi-layer feed-forward networks
3. Recurrent networks

For our purposes, we can treat the first two categories as being the same with single-layer networks being treated a specific case of the more general multi-layer networks. We also focus only on feed-forward networks in this project: recurrent networks are quite different in nature to feed-forward networks and merit separate research.

2.1.1 Feed-Forward Neural Networks

Feed-forward neural networks (FFNNs) are the simplest category of neural network. Their key distinguishing feature is the restriction that there are no cycles in the network. This means the *signal* from the inputs simply moves from the start of the network till the end.

The side-effect of this is that there is no sense of memory in a feed-forward neural network. That is, for a given input and a fixed feed-forward neural network the same output will **always** be produced: the output will not depend on the history of inputs to the network. This is in stark contrast to recurrent neural networks.

Remarkably for a topic within which so much research has been performed, a formalisation of feed-forward neural networks has proved remarkably difficult to find. Many sources give an intuitive definition or omit one altogether relying instead on preexisting knowledge on the part of the reader. The following definition draws inspiration from [26]'s definition of a general neural network:

> **Definition 2.1** (Feed-Forward Neural Network)**.** Let V be a set of vertices, $E \subseteq V \times V$ be a set of edges, $w : E \to \mathbb{R}$ be an **edge weight function** and $b : V \to \mathbb{R}$ be a **bias function**.
>
> A weighted, directed, acyclic graph $N = (V, E, w, b)$ is known as a **feed-forward neural network** (FFNN) if the following properties hold:
>
> 1. For $j = 1...k$, there exists an ordered collection of ordered sets $L^{(j)} \subseteq V$ known

as **layers** such that

$$V = \biguplus_{j=1}^{k} L^{(j)}$$

and for a vertex $v \in L^{(j)}$ $j < k$, the set of endpoints for edges originating from v is equal to some subset $L^{(j+1)}$ and for a vertex $v \in L^{(k)}$ the set of endpoints from v is equal to the empty set.

2. Each $L^{(j)}$, $j \geq 2$ has associated with it a function $\sigma^{(j)} : \mathbb{R} \to \mathbb{R}$ known as the **activation function**.

Remark. *Vertices of a feed-forward neural network are referred to as* ***neurons*** *and edges are referred to as* ***channels****.*

Remark. $L^{(1)}$ *is known as the* ***input layer*** *and* $L^{(k)}$ *is known the* ***output layer****. The remaining* $k - 2$ *layers are collectively referred to as a* ***hidden layers****.*

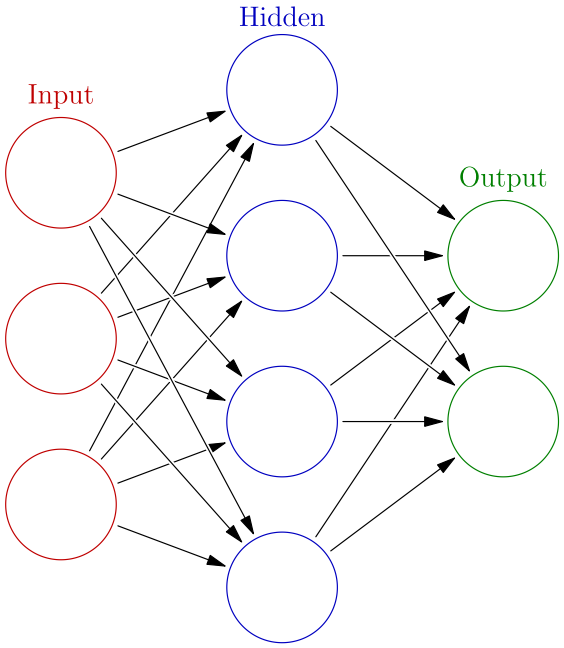


Figure 2.2: Architectural diagram of a feed forward neural network. The vertices represent the neurons and the edges are channels with the weights of the network corresponding to the edges. [1]

Figure 2.2 represents the above, rather complex definition in a diagrammatic way which is much easier to comprehend.

[1] Authored by Glosser.ca

Our work extensively utilises the weights and biases of neural networks so it is useful to define these weights as a matrix and biases as a vector for each layer of the network.

Definition 2.2 (Bias vector of a FFNN layer)**.** Let $N = (V, E, w, b)$ be a FFNN. Then, for each layer $L^{(k)},\ k \geq 2$, we define the **bias vector** to be the vector $b^{(k)} \in R^m$ with $m = |L^{(k)}|$ defined element-wise as:

$$b_i^{(k)} = b(L_i^{(k)})$$

Definition 2.3 (Weight matrix of a FFNN layer)**.** Let $N = (V, E, w, b)$ be a FFNN. Then, for each layer $L^{(k)},\ k \geq 2$ of N, we define the **weight matrix** to be the matrix $W^{(k)} \in R^{m \times n}$ with $m = |L^{(k)}|$ and $n = |L^{(k-1)}|$ defined element-wise as:

$$W_{ij}^{(k)} = w(L_j^{(k-1)}, L_i^{(k)})$$

Intuitively, the above associates a weights matrix to each layer such that (i, j)th entry of the matrix is equal to the weight associated with the corresponding edge between vertex i of the previous layer and vertex j of the current layer.

With these two definitions, we can now define the function computed by each layer of the network. These can then be composed together to give the function computed by the neural network as a whole.

Definition 2.4 (Function computed by single layer)**.** Let $N = (V, E, w, b)$ be a neural network and $2 \leq i \leq k$ with k the number of layers in N.

Then, layer $L^{(i)}$ defines a function $f^{(i)} : \mathbb{R}^m \to \mathbb{R}^n$ known as a **computed function** where $m = |L^{(i-1)}|$, $n = |L^{(i)}|$ and defined as follows:

$$f^{(i)}(x) = \sigma^{(i)}(W^{(i)}x + b^{(i)})$$

Definition 2.5 (Function computed by network)**.** Let $N = (V, E, w, b)$ be a neural network with k the number of layers in N.

Then, N defines a function $f : \mathbb{R}^m \to \mathbb{R}^n$ known as the **computed function** where $m = |L^{(1)}|$, $n = |L^{(k)}|$ and defined as follows:

$$f(x) = (f^{(k)} \circ f^{(k-1)} ... \circ f^{(2)})(x) = f^{(k)}(f^{(k-1)}(...(f^{(2)}(x))))$$

We can see that by defining the weights matrix as a proxy for the weights encoded by the graphical representation of the neural network, we are able to simply perform a matrix-vector multiplication with the weights matrix when computing the output of

each layer.

2.1.2 Training FFNNs

We will briefly discuss some of the concepts of training neural networks. As training is orthogonal to our work on this project, we do not think it is important to have a deep understanding but having the background knowledge may make aid in understanding some of the concepts of the project.

Supervised learning is, at its core, learning by teaching. Humans will collect or generate both the domain and image of a function and then train the network to learn the function mapping between the two.

To train using supervised learning, the neural network is fed two lists of data. The first list is the input values, that is some list X. The second is the list of output values, that is some list $Y = [f(x)|x < -X]$. Essentially, this data is a sample of points for some function f for which the input and output are both known.

The learning process then involves the network trying to minimise the loss function over this data. Most generally the mean-squared error function is used but there are many others which may used.

We will see that supervised learning is used for some of the networks in this project including in some problems in reachability verification and as well as approximating non-linear functions for our work on environment functions.

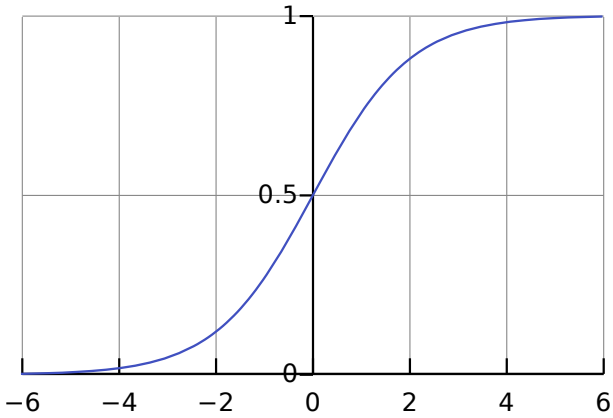
Reinforcement learning is more common in applications where the "correct answer" that a neural network should give is not known. For example, take the game chess. As a human, given an arbitrary chess board, we do not know any method that we could calculate the absolute best move which could be made. We may be able to suggest "good" moves but finding the best one requires us to search the entire future of the board which is infeasible.

This learning process involves the network learning by trying to gain experience by interacting with the environment to maximise some reward function. For example, in the case of chess, the environment would be the board and the reward function may be the number of moves until check mate. The network would play millions of games until it learns a strategy which does "well" according to the reward function.

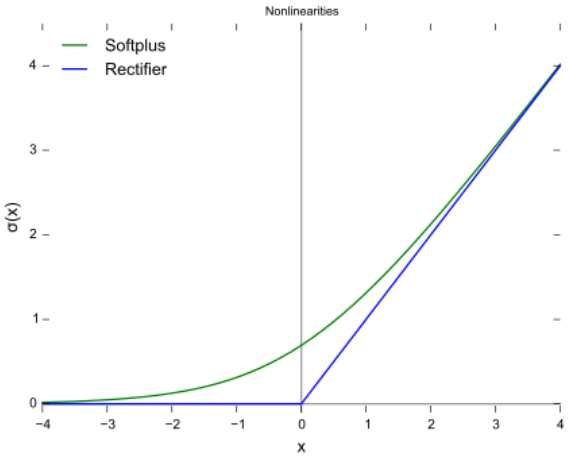
Reinforcement learning is used to generate most of the networks seen in this project.

For example, any agent network in the project was trained using reinforcement learning.

2.1.3 Activation Functions



(a) Graph of the sigmoid activation function. As we can see, the function does not pass through the origin of the graph which induces a bias when training.



(b) Graph of the ReLU function (indicated by the blue line on the graph) as well as an everywhere differentiable approximation to it known as the soft-plus function (indicated by the green line). Soft-plus is also a suitable candidate for use as an activation function but is comparatively rare so has not been discussed.

Figure 2.3: Graphs of sigmoid and ReLU activation functions.

While, in theory, any continuous function can be used as an activation function in neural networks, very few have been found to work well in practice. Some of the most widely used are as follows [26, 44, 33]:

1. **Logistic Sigmoid** Perhaps the oldest activation function which gained traction in multi-layer networks, the sigmoid function is defined as follows:

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$$

   However, due to issues with training (such as slow learning the value of x is large in magnitude and the lack of symmetry around zero), this function is now rarely used in feed-forward networks [33]. However, it is still a vital component in other types of networks (including recurrent ones).

2. **Tanh** A transformation of the Logistic Sigmoid function, the tanh function is defined as follows:

$$f(x) = \tanh(x) = 2\sigma(2x) - 1$$

   As the tanh function is zero centred, this function fixes the issue of bias with the sigmoid function. However, it still suffers from slow learning for large magnitudes of x. Again, its use in feed-forward networks is not common but it too can be found in recurrent networks.

3. **ReLU (Rectified Linear Unit)** A relatively new discovery [44], the ReLU function does not attempt to rescale the input to a finite interval unlike the previous two functions. Instead it simply enforces non-negativity and is defined as follows:

$$f(x) = \max(0, x)$$

   This function has become extremely popular in feed-forward networks in recent years and has been a fundamental cornerstone in the state of the art.

The graphs of the ReLU and sigmoid functions can be found in Figure 2.3.

2.1.4 Keras

Keras is a framework for training and testing neural networks. Written in Python, it exposes an intuitive API to build neural networks in a layer by layer approach while offering a high degree of customisability.

Fundamentally, Keras acts as a wrapper framework: rather than implementing many of the mathematical operations required for neural networks, it instead delegates responsibility to one of two back-ends: Theano and TensorFlow. These libraries are responsible for carrying out the mathematical calculations with Keras abstracting over them with a common API.

Given their low level APIs and focus on offering customisability of the finer points of training and our lack of interest in training neural networks in this project, we will not consider the details of these back-ends.

Keras is one of a countless neural network training frameworks: these range from being very closely coupled with a particular language (such as Matlab's Neural Network Toolbox or DeepLearning4J for the JVM) to ones which have bindings for a wide range of languages (such as TensorFlow which has official bindings for Python and C++ as well as unofficial ones for Java and Go). After much consideration of the advantages and disadvantages of many of these options, Keras was settled upon as the framework of choice.

Python has long been known as a friendly language for prototyping tools. As well as this, it has long been used by researchers in Machine Learning to carry out experiments and build tools. As this is precisely what we are trying to carry out in this project, the support from documentation and the community would be quite extensive compared to another language.

Within Python itself again many options are available. However, none is as mature or as well regarded as Keras. From a more subjective point of view, we found that the API of Keras was a pleasure to use: it gave us easy access to weights and biases for example which many other frameworks had much more convoluted means to access. It also was easy to set up training in a sensible manner without much customisation as we were not interested in this aspect of neural networks.

2.2 Linear Programming

Linear programming is an technique devised by the field of operations research. It is a specific case of mathematical optimisation where the function we are attempting to optimise (i.e. the objective function) and the constraints on optimisation are both linear. With this restriction, efficient algorithms have been devised which are able to perform this optimisation.

The following definitions are adapted from [61].

Definition 2.6 (Linear Function)**.** A function $f : \mathbb{R}^N \to \mathbb{R}$ is said to be **linear** if and only if for some $c \in \mathbb{R}^N$, we have $f(x) = \sum_{i=1}^{N} c_i x_i$

Definition 2.7 (Linear (in)equality)**.** For any linear function $f : \mathbb{R}^N \to \mathbb{R}$, and any $b \in \mathbb{R}$, $f(x) = b$ is said to be a **linear equality** and $f(x) \leq b$ and $f(x) \geq b$ are said to be **linear inequalities**.

Definition 2.8 (Linear Program)**.** A **linear program** (LP) is a mathematical optimisation problem where the following conditions hold:

1. The function we are optimising is linear. This function is known as the **objective function**.
2. The variables of the optimisation are constrained with every constraint being linear equality or inequality.
3. The sign of every decision variable restricted: the variable is required to be non-negative or unconstrained.

These definitions lay the foundations of linear programming. We now consider the addition of integer variables which allows for an greater number of classes of problems to be solved.

Definition 2.9 (Mixed Integer Linear Programs)**.** A **mixed integer linear program** (MILP) is a linear program which allows for constraints which require variables to be integer. That is constraints of the type $x_i \in \mathbb{Z}$.

An important to note is that while polynomial time algorithms exist for the solution for real-valued linear programs, the addition of integer variables changes the problem to one which is NP-complete. This means there does not exist an algorithm for the solution of these problems which is polynomial time for all inputs. Moreover, this property means that, for each algorithm certain classes of MILPs are likely to be easier to solve than other classes which emphasises the importance of which encoding of a neural network we pick.

Gurobi is a mathematical optimisation software which has the ability to solve linear programs, MILPs and other, more complex, problems. It is a commercial software but is free for academic use (i.e. all work carried out by this project).

Gurobi is by no means the only linear program solver which could be used; as well as free

software such as GNU GPLK and CBC, there exist a vast array of commercial solvers such as CPLEX, SCIP and Gurobi itself.

However, as we will be solving large linear programs when performing verification speed was a very important factor. When we attempted to research into benchmarks for these problems, we found that the de facto one is the Mittelmann benchmark.

This benchmark (created by an academic Hans Mittelmann), tests all solvers on a wide range of problem sets and reports the results of these tests. When the results of these tests are considered, we find that Gurobi is a very strong performer in the linear programming category coming second and takes a strong first place when it comes to MILPs.

As well as its speed, Gurobi has an API which can be directly used by a wide range of language including our choice of Python. This API was found to be relatively intuitive and flexible.

For these reasons, Gurobi was chosen as our solver of choice. However, it would be prudent to investigate the performance of other solvers on the techniques used in this project as it may be the case that the structure of the problems built are better solved in the future. Due to time constraints, we have not managed to carry out this task in the project.

2.3 Systems Verification

Systems Verification is a field of Computer Science which develops techniques to check the correctness of some component of a computer program.

Every computer system has an expected output behaviour: this is its specification. Usually this is an informal description of the program is supposed to produce for normal inputs and how it should react to exceptional behaviour. For critical applications, such an informal description may not be sufficient - it may be required to have a more formal description of the behaviour. Verification is the process of taking a formal specification and then checking if the system satisfies the specification.

To get a sense for the components of the process of verification, suppose that we have a system and a specification which we wish to check that the system satisfies. At a high level, verification involves the following [31]:

1. A *framework* to model the system making it amenable to perform verification.

2. A language with which we can encode the specification. This is know as the *specification language*.

3. A technique which can check if the system satisfies the specification. This is known as a *verification technique*.

There are clearly may possible approaches to verification we could take. It is useful to classify them using the following criteria [31]:

- **Proof-based or model-based** Proof based techniques encode the system as a set of formulae (denoted Γ) and the specification being verified as another formula (denoted ϕ) and then attempt to find a proof for $\Gamma \vdash \phi$.

  In a model based technique, a model of the system is built (denoted M) and the specification is again encoded as a formula (again ϕ) and then a check is carried out to see if $M \vDash \phi$.

- **Degree of automation** The level of automation of the proof techniques varies wildly from completely automatic to completely manual.

- **Full- or property-verification** The specification could describe the behaviour of the system under all conditions or simply denote a certain important property the system should have.

  Clearly the former method is a very expensive process: both in terms of creating the specification as well as actually performing verification. Therefore, usually, certain important properties (e.g. fairness - that a system eventually responds to every input - or liveness - that a system always makes progress i.e. never deadlocks) are checked instead.

There are two main techniques we will draw inspiration from when performing verification on neural networks: these are *model checking* and *static program verification*.

2.3.1 Model Checking

Model checking is a *lightweight* verification method. Using the categorisation described above, model checking is a model based, fully automatic, property based verification technique.

Model checking accepts that most systems are far too complex to verify directly. Instead, for a system S for which we are attempting to show that a property P holds, we build a

model for the system M_S and encode the property P as a logical formula ϕ_P. We then check using proof techniques whether $M_S \vDash \phi_P$.

The exact details of model checking depends heavily on the the specification language with which properties are encoded. We focus on one specifically in this project: *Linear Temporal Logic*.

Linear Temporal Logic (LTL)

When working with network systems, it is not enough to simply have mathematical constraints on the inputs or outputs of the networks when encoding safety properties for the network due to temporal component of these networks.

Linear Temporal Logic (LTL) is a specification language which is ideal for modelling precisely this sort of behaviour. The language breaks up time into discrete events or states and defines the concept of paths over these states.

Definition 2.10 (LTL Syntax)**.** Let p be any propositional atom. Then the syntax of the LTL language is defined as follows:

$$\phi, \psi ::= \top \mid p \mid \neg\phi \mid \phi \wedge \phi \mid X\phi \mid \phi U \psi$$

Remark. *We take further the following abbreviations:*

- $F\phi ::= \top U \phi$
- $G\phi ::= \neg F \neg \phi$
- $\phi R \psi ::= \neg(\neg\phi U \neg\psi)$

Before we define the semantics of LTL languages, we need to define the concept of *models* and *paths*. These are the structures over which we will interpret LTL formulae. We assume the reader is familiar with the concept of Kripke models which we will not define.

Definition 2.11 (LTL Models)**.** Let $M = (W, R, \pi)$ be a Kripke Model. We say that M is a **model** for LTL if R is a serial relation.

Remark. *R is said to be a serial relation if any state is related to at least one other state.*

Definition 2.12 (LTL Paths)**.** Let $M = (W, R, \pi)$ be a LTL Model. A **path** ρ for this model is any infinite sequence $w_0, w_1, ...$ such that $(w_i, w_{i+1}) \in R$, for any $i \geq 0$.

Remark. *We will use the notation ρ^i to refer to a suffix of ρ i.e. the path starting at w_i, $\rho^i = w_i, w_{i+1}, ...$*

Now that we have these preliminary definitions, we can define satisfaction of LTL formulae both on a specific path and on worlds of the model.

Definition 2.13 (LTL Satisfaction on Path)**.** Let $M = (W, R, \pi)$ be a LTL Model, ϕ be an LTL formula and $\rho = w_0, w_1, ...$ be an LTL path on M. **Satisfaction** of ϕ on ρ is defined as follows:

$$
\begin{array}{lll}
(M, \rho) \vDash \top & & \\
(M, \rho) \vDash p & \text{iff} & w_0 \in \pi(p) \\
(M, \rho) \vDash \neg\phi & \text{iff} & (M, \rho) \nvDash \phi \\
(M, \rho) \vDash \phi \wedge \psi & \text{iff} & (M, \rho) \vDash \phi \text{ and } (M, \rho) \vDash \psi \\
(M, \rho) \vDash X\phi & \text{iff} & (M, \rho^1) \vDash \phi \\
(M, \rho) \vDash \phi U \psi & \text{iff} & \exists j \geq 0 \text{ such that } (M, \rho^j) \vDash \psi \text{ and} \\
 & & \forall 0 \geq k < j \text{ we have } (M, \rho^k) \vDash \phi
\end{array}
$$

Definition 2.14 (LTL Satisfaction at State)**.** Let $M = (W, R, \pi)$ be a LTL Model, ϕ be an LTL formula and w be a world of M. We say ϕ is true at or **satisfied** at w written $(M, w) \vDash \phi$, if for all paths ρ starting at w, we have $(M, \rho) \vDash \phi$

While there is much more to be said about LTL in general, these definitions have introduced all the concepts we require for this project.

2.3.2 Static Program Verification

Static program verification is an alternative verification method used in the field of systems verification. From our earlier categorisation, this technique would be proof-based, fully automatic, property verification.

In contrast to model checking which generates an abstract model of a program before proving a specification on this model, static program verification works directly upon the program itself.

Generally a static program verifier takes as input the program which is attempting to be verified as well as the specification attempting to be verified in the form of a first order formula [6]. From these two components, it generates a *verification condition*. This is a first order formula which, if valid, implies that the program is correct with respect the specification.

This formula is then passed to some sort of theorem prover. Alternatively, instead of checking the validity of the formula, one can check that the negation of the specification is unsatisfiable: in this case, the verification condition can instead be passed to an SMT solver which will check the satisfiability of the formula: if this formula is unsatisfiable, then the program is assumed to satisfy the specification and vice-versa.

More formally, static program verification involves reasoning about the *partial correctness* of the program which is defined as follows [13]:

> **Definition 2.15.** Suppose P is a first order formula (known as the **pre-condition**) and Q another first order formula (known as the **post-condition**). Suppose further we have a program (system) S.
>
> Then we say the S is partially correct, if for every the starting state of the machine which satisfies formula P one of the following occurs:
>
> - The program S terminates and the final state of the system satisfies the formula Q.
> - The program S does not terminate.

Of course, the difficult part of this process is actually translating the system and specification to a verification condition. In general, due to the link to the halting problem, it is impossible to carry out this process exactly to fully verify the system.

The general trade-off made is to *over-approximate* the system which is being verified [13]. Instead of generating a verification condition which perfectly models the system, one creates a condition which captures the behaviour of the program *and possibly other impossible behaviours*.

We will will take inspiration from some of these concepts when we consider temporal verification.

2.4 Bridging the gap

In the introduction we have mentioned that very little research has been done in the area of intersection between systems verification and neural networks. Here, we review the research to date.

Safety for neural network was first discussed in a form quite applicable to this project in [35] and was expanded upon by the same authors in [37]. The main insight we can gain from this report involves the different types of safety properties which may be important to check on neural networks. Specifically, the authors say that the following properties are important to check:

1. The neural network accurately represents the function which the network is attempting to approximate.

2. The outputs of neural networks are repeatable and can be predicted in advance.

3. The neural network can tolerate inputs which are invalid instead of failing.

4. The output of neural networks are always safe.

These definitions are very informal but still form a good starting point for verify safety proprties for neural networks

In particular, we feel that the third and fourth properties are especially important; the first two properties are much more abstract properties which are difficult to check. The third and fourth are results which can be proved formally using verification techniques and thus will be the focus of this project.

This first paper we have found which takes a more formal approach to verification is given by [46]. In this paper, the authors describe a technique for verifying safety properties of feed-forward neural networks using a technique of "abstraction". The safety property they attempt to verify is checking that for every possible input value, the output is within a certain bounded interval.

The technique of abstraction they propose is one of gradually refining an approximation of a neural network using SMT constraints and then feeding this to a SMT solver to attempt to find a counter-example input which maps outside the "safe" region. If the real network does not map this input in the same way, they refine their approximation by adding constraints and repeat this process until both the real network and the approximation agree that all inputs are safe or both agree on a counter-example.

This approach is demonstrated by the authors on a network only 6 neurons big which left questions as to the scalability of the approach and was the focus a follow up paper published by the same authors [47]. Here, they admit that the technique of using SMT solvers is not one which is particularly scalable and there were large challenges to be overcome.

A very similar approach was also proposed in [55] which instead calls this technique bounded model checking and moved this process within the SMT solver itself. However, it too concluded that the scalability was lacking.

The other techniques we have found attempt to find so called "adversarial examples". This technique works in particular on classification neural networks - networks which divide the inputs of the network into a finite number of discrete classes.

An adversarial example for a particular input to the network, x, is an input x^* which is, with respect to some norm, close to x but has a different classification given by the neural network.

One such work is [8]. Here the authors encode neural networks as linear programs to find such examples. Specifically, for an input x they attempt to find an example which has a different classification and is the smallest distance away from x. This is an optimisation problem and, through some partitioning of the input space (and restricting the network layers' activation functions to be ReLU functions), they encode this as a linear program which can be solved efficiently.

This technique has a high potential as the authors demonstrate it working on large problems. Moreover, linear programs are generally (but not always) much more efficient to solve than SMT problems as SMT is NP-Complete but linear programming is PTIME. In fact, while this paper was not known about when developing the LP techniques used in the implementation of the project, it was later found that the encoding of neural networks was quite similar between our project and this paper.

Another paper we feel is of relevance is [30]. Here, the authors utilise a layer by layer approach using activations of networks directly rather than working with activation functions. They then attempt to generate a covering set around the input using "manipulations" - small perturbations which can be made to the image. Then these are exhaustively searched using an SMT solver to see if any of these examples could lead to a misclassification.

The scalability of this technique is extremely good as it operates in a layer by layer fashion. Moreover, the authors demonstrate this technique working on a realistically

large neural network which achieved close to state-of-the-art performance on image classification. However, this technique is very specific to adverserial examples and is not one which can be applied to reachability.

Recently, another paper, which has similar aims of verifying reachability, was recently made available on ArXiv [19]. However, there is no formal correspondence presented between reachability analysis and linear programs as we do here. Moreover, the underlying techniques proposed are different.

While their method is based on SMT-solving, we only use linear programming here. Linear programming is used in [19] as a comparison against SMT, but there is no mention of any optimisation on the LP engine. In contrast, we here focus on an efficient LP translation and handle floating point operations in an optimised manner. Also the scenarios studied are different from ours and have not been released to allow for a comparison.

Judging from the experimental results presented for the LP comparison, the LP technique used in [19] performs significantly less efficiently than what we develop here. For example, their analysis is only shown with up to 300 ReLU constraints whereas our technique is able to handle more than 500 with ease. An in-depth comparison of the performance of the two different techniques will not be possible until all data in [19] are released.

Even more recently, yet another people appeared on ArXiv which also attempted to verify reachability for feed-forward ReLU networks [17]. This technique is somewhat of a hybrid one between SAT and linear programming. Instead of using only a SAT or LP solver, the technique uses both. SAT is used as the core of the algorithm with traditional techniques such as unit propagation and conflict detection being used from here. However, feasibility of a solution is also checked with a linear programming solver.

While very interesting, we have not had time to get a good sense of exactly how this paper operates due to its release late in the project cycle. We believe that given the size of networks they evaluated, that our technique would be comparable in performance.

Relating each of these papers back to the informal definitions of safety proposed in [37], we see that [47] and [55] are attempting to verify the first safety property. While this is perhaps the broadest safety property, it intuitively feels very hard to demonstrate and this is backed up by the lack of scalability of the papers

[8] and [30] which instead work on adversarial examples are in some sense attempting to prove a safety property somewhere between the third and fourth types of safety. While adversarial examples are not exactly faulty inputs in general, they can be considered

a fault with respect to a specific category. They are also related to the fourth type in some sense as miscategorising the output of the adversarial example is in some sense generating an output which is not safe.

[19] and [17] are also somewhere between the third and fourth types of safety. Reachability can be used to ensure that certain outputs are never produced or that a portion of inputs are always mapped to certain outputs. This closely reflects the intuitive meaning behind both properties three and four.

In summary, we have given a overview of the different attempts there have been from bringing verification of safety properties to neural networks. We believe that all these papers suffer for a drawback, have a different aim or use different techniques to the ones we use. We also note that all the work has focused on one-step systems (i.e. reachability or adversarial examples). Not one paper has attempted to introduce a temporal element to their analysis.

3 Reachbility for feed-forward neural networks

In this section, we discuss the topic of reachability and how it can be verified for neural networks.

As with most of the work in the project, all the work in this chapter is essentially novel. We start by defining the very concept of reachability itself, something which we has not been defined to date. We then consider an encoding of a neural network as a set of linear constraints. While similar encodings have been presented in the past, our one is more compact, efficient and uses fewer variables.

We then discuss how we can use this encoding in defining a linear program which, when solved, can decide whether a set is reachable from another through the neural network. Our formality throughout in proving the links between reachability and the solution to the LP mean that we are able to show that the final technique is both sound and complete which is a strong result which no other similar algorithm has proved to date. Finally, we finish by presenting a sound and complete algorithm which uses the above technique but also handles floating point considerations; we believe that this is the first reachability verification algorithm to make this claim.

3.1 Defining Reachability

As discussed in the background, reachability is fundamentally about defining whether certain set of target states is "connected" (usually by a path) to a certain set of starting states. To date, no definitions have been given for how reachability might work for a feed-forward neural network. In this section, we seek to create such a definition.

We first define the notion of a "set of states" for FFNNs. A reasonably immediate definition follows by observing that, fundamentally, a neural network computes a function: this function was defined in Chapter 2. As the function has a well defined domain and range, this can be used as a basis for as our definition.

Definition 3.1 (State spaces of FFNNs)**.** Let N be a neural network with computed function $f : R^m \to R^n$ where $m = |L^{(1)}|$ and $n = |L^{(k)}|$ where k is the number of layers in N.

Then, for any $I \subseteq R^m$, I is an input or *starting state set* for N and for any $O \subseteq R^n$, O is an output or *target state set* for N.

With these definitions of state spaces, we can now define the notion of reachability for feed-forward neural networks.

Definition 3.2 (Reachability for FFNN)**.** Suppose N is an FFNN with computed function $f : \mathbb{R}^m \to \mathbb{R}^n$ with $m = |L^{(1)}|$, $n = |L^{(k)}|$, where k is the number of layers in N. Let $I \subseteq \mathbb{R}^m$ be a starting state set and $O \subseteq \mathbb{R}^n$ be a target state set.

We say that O is *reachable from* I through N if $\exists y \in O, \exists x \in I, f(x) = y$.

We believe this is a definition which captures many of the properties that one may require for a definition of reachability: it defines very clear notions of what the starting and target sets are, by which mechanism they are related and when given an example of a $x \in I$ and $y \in O$, is able to easily determine whether y is reachable from x by simply calculating the value of a function.

However, this definition is not without problems for developing a practical verification algorithm. The main issue is that the input and output sets are allowed to be arbitrary subsets of the real vector spaces R^m and R^n. This allows for sets like $I = \{x \mid x \text{ an irrational number}\}$. While they can be clearly defined in English, these sets are infinite and it is impossible to enumerate all their members programmatically.

A further distinction needs to be made with sets which can be enumerated but which are likely to be difficult to perform verification over. These include sets such as $I = \{x \mid P(x) = 0\}$ where P is a second order (or higher) polynomial.

For this reason, we impose strict restrictions on exactly how the starting and target sets are defined. As we seek to perform verification by drawing an equivalence between reachability and solving linear programs, an immediate idea is to restrict the definition of these sets to linear equalities and inequalities.

We now formally define the exact state sets that we will perform verification on in this project.

Definition 3.3 (Linearly Definable Set.)**.** Let $S \subseteq \mathbb{R}^n$. We say that S is linearly

definable if and only if there exists a finite set of linear constraints C_S such that $S = \{x \in \mathbb{R}^n \mid x \text{ satifies every constraint in } C_S\}$. We define C_S to be the constraint set of S.

We note that this still enables us to capture a large number of systems since all linear equalities and disequalities are allowed. This includes verification of ACAS networks performed on in [19] as both the starting and target sets in these problems are linearly definable as well as the wide variety of problems we will solve and experiments we perform later in this chapter.

With this definition, we now state the more restricted variant of reachability which we aim to verify in this project.

Definition 3.4 (Restricted Reachability for FFNN)**.** Suppose N is an FFNN with computed function $f : \mathbb{R}^m \to \mathbb{R}^n$ with $m = |L^{(1)}|$, $n = |L^{(k)}|$, where k is the number of layers in N. Let $I \subseteq \mathbb{R}^m$ be a linearly definable starting state set and $O \subseteq \mathbb{R}^n$ be a linearly definable target state set.

We say that O is *reachable* from I through N if $\exists y \in O, \exists x \in I, f(x) = y$.

3.2 Encoding neural networks using linear inequalities

We now show that a neural network with layers which have either ReLU or identity activation functions can be represented using a set of linear constraints.

While informal linear encodings for individual neurons have been proposed in the past [19], we instead propose an encoding which works on a layer-by-layer manner. Moreover, our ReLU encoding utilises only a single binary variable; as we demonstrate later this is important for efficiency in practical applications.

We first propose the obvious encoding for layers with identity activations.

Definition 3.5 (Linear encoding for identity FFNN layer)**.** Let N be an FFNN and $2 \leq i \leq k$ where layer i uses an identity activation function and k is the number of layers in N. Let W be the weights matrix and b the bias vector of N.

Suppose further $x^{(i-1)}$ and $x^{(i)}$ are vectors of real LP variables representing the input and output of layer i respectively. Then, the set of *linear constraints encoding layer i*

(with an identity activation function) is defined as:

$$C_i = \{x_j^{(i)} = W_j^{(i)} x^{(i-1)} + b_j^{(i)} \mid j = 1...|L^{(i)}|\}$$

We now define a similar, but more complex, encoding for layers which use a ReLU activation function.

Definition 3.6 (Linear encoding for ReLU FFNN layer)**.** Let N be an FFNN and $2 \leq i \leq k$ where layer i uses a ReLU activation function and k is the number of layers in N. Let W be the weights matrix and b the bias vector of N.

Suppose further $x^{(i-1)}$ and $x^{(i)}$ are vectors of real LP variables representing the input and output of layer i respectively and $\delta^{(i)}$ a vector of binary LP variables. Then, the set of *linear constraints encoding layer* i, (with a ReLU activation function) is defined as:

$$\begin{aligned} C_i = \{&x_j^{(i)} \geq W_j^{(i)} x^{(i-1)} + b_j^{(i)}, \\ &x_j^{(i)} \leq W_j^{(i)} x^{(i-1)} + b_j^{(i)} + M\delta_j^{(i)}, \\ &x_j^{(i)} \geq 0, \\ &x_j^{(i)} \leq M(1 - \delta_j^{(i)}) \mid j = 1...|L^{(i)}|\} \end{aligned}$$

where M a "sufficiently large" constant.

Next, we show that input and output vectors which satisfy the constraint set for a layer are precisely those related by the computed function of the layer.

Proposition 3.1 (Linear encoding for ReLU layer equivalent to computed function)**.** *Let N be an FFNN and $2 \leq i \leq k$ where layer i uses a identity or ReLU activation function and k is the number of layers in N. Let C_i be the constraint set encoding layer i and $f^{(i)} : \mathbb{R}^m \to \mathbb{R}^n$ be the computed function of layer i where $m = |L^{(i-1)}|$ and $n = |L^{(i)}|$.*

Then $\forall \mathbf{x}^{(i-1)} \in \mathbb{R}^m, \mathbf{x}^{(i)} \in \mathbb{R}^n$, we have $\mathbf{x}^{(i)} = f^{(i)}(\mathbf{x}^{(i-1)})$ if and only if $\exists \boldsymbol{\delta}^{(i)} \in \{0,1\}^n$ such that $\mathbf{x}^{(i-1)}$ $\mathbf{x}^{(i)}$ and $\boldsymbol{\delta}^{(i)}$ (substituted for $x^{(i-1)}$, $x^{(i)}$ and $\delta^{(i)}$ respectively) satisfy every constraint in C_i.

Proof. For layers using identity activation, the proof is immediate upon inspection of Definitions 2.4 and 3.5 and utilising any binary vector for $\boldsymbol{\delta}^{(i)}$. □

Proof. For layers using ReLU activation activation, we must consider each direction in turn. For conciseness, we use the short-form notation $\mathbf{y}_j \equiv W_j^{(i)} \mathbf{x}^{(i-1)} + b_j^{(i)}$.

$\implies$ We have that $\mathbf{x}_j^{(i)} = \max(\mathbf{y}_j, 0)$ for $j = 1...n$. Then certainly, by definition, we

have that $\mathbf{x}_j^{(i)} \geq 0$ and $\mathbf{x}_j^{(i)} \geq \mathbf{y}_j$. Moreover, either $\mathbf{y}_j > 0$ or $\mathbf{y}_j \leq 0$.

Suppose $\mathbf{y}_j > 0$. We have that $\mathbf{x}_j^{(i)} = \mathbf{y}_j$. Taking $\boldsymbol{\delta}_j^{(i)} = 0$, we certainly satisfy the constraint $\mathbf{x}_j^{(i)} \leq \mathbf{y}_j + M\boldsymbol{\delta}_j^{(i)}$ and, for sufficiently large M, we also satisfy the constraint $\mathbf{x}_j^{(i)} \leq M(1 - \boldsymbol{\delta}_j^{(i)})$ as required.

Suppose $\mathbf{y}_j \leq 0$. We have that $\mathbf{x}_j^{(i)} = 0$. Taking $\boldsymbol{\delta}_j^{(i)} = 1$, we certainly satisfy the constraint $\mathbf{x}_j^{(i)} \leq M(1 - \boldsymbol{\delta}_j^{(i)})$ and, for sufficiently large M, we also satisfy the constraint $\mathbf{x}_j^{(i)} \leq \mathbf{y}_j + M\boldsymbol{\delta}_j^{(i)}$ as required.

Therefore, substituting the vectors $\mathbf{x}^{(i-1)}$, $\mathbf{x}^{(i)}$ and $\boldsymbol{\delta}^{(i)}$ for the equivalent LP variables in C_i, we see that all the constraints in C_i are satisfied.

$\Longleftarrow$ We have that $\exists \boldsymbol{\delta}^{(i)}$ such that $\mathbf{x}^{(i-1)}$, $\mathbf{x}^{(i)}$ and $\boldsymbol{\delta}^{(i)}$ substituted for the equivalent LP variables satisfy all the constraints in C_i. We have further that $\boldsymbol{\delta}_j^{(i)}$ is either equal to 0 or 1 as $\delta_j^{(i)}$ is a binary LP variable for $j = 1...n$.

Suppose $\boldsymbol{\delta}_j^{(i)} = 0$. We have from C_i that $\mathbf{x}_j^{(i)} \geq \mathbf{y}_j$ and $\mathbf{x}_j^{(i)} \leq \mathbf{y}_j + M\boldsymbol{\delta}_j^{(i)} = \mathbf{y}_j$ which implies $\mathbf{x}_j^{(i)} = \mathbf{y}_j$. Moreover, we have that $\mathbf{y}_j = \mathbf{x}_j^{(i)} \geq 0$.

Suppose $\boldsymbol{\delta}_j^{(i)} = 1$. We have from C_i that $\mathbf{x}_j^{(i)} \geq 0$ and $\mathbf{x}_j^{(i)} \leq M(1 - \boldsymbol{\delta}_j^{(i)}) = 0$ which implies $\mathbf{x}_j^{(i)} = 0$. Moreover, we have that $0 = \mathbf{x}_j^{(i)} \geq \mathbf{y}_j$.

Therefore, in either case, we have that $\mathbf{x}_j^{(i)} = \max(\mathbf{y}_j, 0) = f^{(i)}(\mathbf{x}_j^{(i-1)})$ as required. □

Finally, we define the constraint set for the network as a whole and show that the input and output vectors for the network are also precisely related by the computed function of the network.

Definition 3.7 (Linear encoding for FFNN)**.** Let N be an FFNN with every layer of N using either a ReLU or linear activation function.

Then we have that set of *linear constraints encoding the network* is $C = \cup_{i=2}^{k} C_i$ where C_i is the set of linear constraints encoding layer i.

Theorem 3.1 (Linear encoding for FFNN equivalent to computed function)**.** *Let N be an FFNN and C be the set of linear constraints encoding N. Let $f : \mathbb{R}^m \to \mathbb{R}^n$ be the computed function of layer i where $m = |L^{(1)}|$ and $n = |L^{(k)}|$.*

Then $\forall \mathbf{x}^{(1)} \in \mathbb{R}^m, \mathbf{x}^{(k)} \in \mathbb{R}^n$, we have $\mathbf{x}^{(k)} = f(\mathbf{x}^{(1)})$ if and only if $\exists \mathbf{x}^{(i)} \in \mathbb{R}^{|L^{(i)}|}, i = 2...k-1$ and $\exists \boldsymbol{\delta}^{(i)} \in \{0,1\}^{|L^{(i)}|}, i = 2...k$ such that $\mathbf{x}^{(1)}$, $\mathbf{x}^{(i)}$, $\mathbf{x}^{(k)}$ and $\boldsymbol{\delta}^{(i)}$ (substituted

for $x^{(1)}$, $x^{(i)}$, $x^{(k)}$ and $\delta^{(i)}$ respectively) satisfy every constraint in C.

Proof. Note, by definition of f, we have that $f(\mathbf{x}^{(1)}) = f^{(k)}(f^{(k-1)}(...f^{(2)}(\mathbf{x}^{(1)})))$.

We consider each direction in turn.

$\implies$ Define $\mathbf{x}^{(i)} := f^{(i)}(...f^{(2)}(\mathbf{x}^{(1)})) = f^{(i)}(\mathbf{x}^{(i-1)})$ for $i = 2...k-1$.

Now by Proposition 3.1, we have $\exists \boldsymbol{\delta}^{(i)}$ such that the substitution of the $\mathbf{x}^{(i-1)}$, $\mathbf{x}^{(i)}$ and $\boldsymbol{\delta}^{(i)}$ into C_i satisfies all the constraints in C_i.

Since $C = \cup_{i=2}^{k} C_i$, by using substituting these $\mathbf{x}^{(i)}$ and $\boldsymbol{\delta}^{(i)}$ into C, all constraints in C are satisfied.

$\impliedby$ Since $\mathbf{x}^{(i-1)}$, $\mathbf{x}^{(i)}$ and $\boldsymbol{\delta}^{(i)}$ satisfy all constraints in C, we have by definition of C that these vectors substituted into C_i must satisfy all constraints in C_i.

Now by Proposition 3.1, we have that $\mathbf{x}^{(i)} = f^{(i)}(\mathbf{x}^{(i-1)}), 2 \leq i \leq k$. Therefore, by repeated application we have $\mathbf{x}^{(k)} = f^{(k)}(...f^{(2)}(\mathbf{x}^{(1)})) = f(\mathbf{x}^{(1)})$ as required. □

Before we finish this section and move onto talking about how this LP can be used to solve reachbility problems, we will take a brief detour to discuss another way we could have encoded a layer of FFNNs; that is by using SOS constraints.

Evidence from literature suggests that encoding piecewise-linear functions (i.e. the ReLU function in our case) using SOS constraints may be more efficient than using just binary variables as we have done. Unfortunately, for most problems, based on some very rough experiments (which we have not detailed in the report), we found this encoding to be less efficient than the one given by Definition 3.6. However, for the MNIST case-study we present later, this encoding was actually found to be *more* efficient so meaningful conclusions are hard to draw.

If we were to use this encoding, we replace the one given by Definition 3.6 with the following; all subsequent definitions and theorems would apply from then onwards would apply without any further modifications.

$$\begin{aligned} C_i = \{&x_j^{(i+1)} = W_j^{(i)} x^{(i)} + b_j^{(i)} + s_{j1}^{(i)}, \\ &x_j^{(i+1)} = s_{j2}^{(i)}, z_{j1}^{(i)} + z_{j2}^{(i)} = 1, \\ &SOS1(s_{j1}^{(i)}, z_{j1}^{(i)}), SOS1(s_{j2}^{(i)} x, z_{j2}^{(i)}) \mid j = 1...|L^{(i)}|\} \end{aligned}$$

where $z_{j1}^{(i)}$ and $z_{j2}^{(i)}$ are binary variables and $s_{j1}^{(i)}$ and $s_{j2}^{(i)}$ are slack variables.

In this project, we were unable to perform a through examination of the trade-offs of this encoding due to time constraints and thus we will not be discussing this encoding any further. However, we strongly believe this is an area which merits further investigation and thus we have included it here.

3.3 Reachbility of neural networks by solving linear programs

Using the encoding defined in the previous section, we give a linear program which captures the essence of restricted reachability.

Definition 3.8 (LP encoding reachability for FFNN.)**.** Let N be an FFNN, C its encoding as per Definition 3.7, and $I \subseteq \mathbb{R}^m$ (respectively, $O \subseteq \mathbb{R}^n$) be a linearly definable set encoding the starting states (target states, respectively) where $m = |L^{(1)}|$ and $n = |L^{(k)}|$.

The *linear program encoding the reachability* of O from I through N is given by the objective function $z = 0$ and constraints $C_{reach} = C_{in} \cup C \cup C_{out}$, with the sign of the variables unconstrained, where

- C_{in} is a constraint set for I defined on the same variables used in the encoding of the input to the second layer of N (i.e. $x_j^{(1)}$ for $j = 1...m$), and
- C_{out} is a constraint set for O defined on the same variables used in the encoding of the output of the last layer of N (i.e. $x_j^{(k)}$ for $j = 1...n$).

We now prove that solving this LP provides a technique to decide restricted reachbility for feed-forward neural networks.

Theorem 3.2 (Equivalence between reachability analysis and corresponding LP problems.)**.** *Suppose N is an FFNN on linearly definable input I and output O. Let L be the corresponding linear problem encoding the reachability of O from I through N (by Definition 3.8).*

Then, O is reachable from I through N if and only if the linear program L has a feasible solution $(\mathbf{x}, \boldsymbol{\delta})$.

Proof. We consider each direction in turn.

$\implies$ We have $\exists x \in I, \exists y \in O$ with $f(x) = y$.

Clearly since $x \in I$, we must have that, by definition, x substituted satisfies all the

constraints in C_{in}. Similarly $y \in O$ satisfies all the constraints in C_{out}.

Furthermore, letting $\mathbf{x}^{(1)} = x$ and $\mathbf{x}^{(k)} = y$, by Theorem 3.1, we have that $\exists \mathbf{x}^{(i)} \in \mathbb{R}^{|L^{(i)}|}$ and $\exists \boldsymbol{\delta}^{(i)} \in \{0,1\}^{|L^{(i)}|}$ such that $\mathbf{x}^{(1)}$, $\mathbf{x}^{(i)}$, $\mathbf{x}^{(k)}$ and $\boldsymbol{\delta}^{(i)}$ substituted satisfy every constraint in C.

Finally, consider the assignment of LP variables $x^{(i)} \to \mathbf{x}^{(i)}$ and $\delta^{(i)} \to \boldsymbol{\delta}_j^{(i)}$. This assignment is a feasible solution as the assignment satisfies allows satisfaction of constraints in C_{reach}.

$\Longleftarrow$ We have $(\mathbf{x}, \boldsymbol{\delta})$ is a feasible solution for L. In a slight abuse of notation, we index into this solution using the familiar notation we have used until this point.

Let $x_j = \mathbf{x}_j^{(1)}$ and $y_j = \mathbf{x}_j^{(k)}$. As $\mathbf{x}^{(1)}$ is part of the the feasible solution, it must have satisfied all constraints in C_{in}. Therefore, by definition, we must have that $\mathbf{x}^{(1)} = x \in I$. Using similar reasoning using C_{out}, we must have that $y \in O$.

Finally, we have that since $(\mathbf{x}, \boldsymbol{\delta})$ are feasible, they satisfy all constraints in C. Therefore, by Theorem 3.1, $y = f(x)$ as required. □

3.4 Transitioning to a reachability verification algorithm

In this section, we will move from the very theoretical definitions we presented above to a practical algorithm which we will implement to verify real-world properties in our section on experiments.

Before, we present our final reachability algorithm, we need to discuss one final issue of floating point errors. This issue is subtle one which requires careful handling to preserve soundness. In short, the problem is due to our encoding of FFNNs being defined on the real number line.

However, computers do not work directly with real numbers as any irrational real number would require an infinite amount of memory to fully represent. Instead they work with an approximation known as *floating point*. The details of exactly how this approximation works is not crucial so we do not include it in this report.

However, the important takeaway for our work is that equalities in floating point are *not* exact. Instead, we need to compare values with some *tolerance*. Therefore, for the encoding to be correct, the constraints present in the resulting LP must take into account a safe level of floating point precision and use tolerances when defining the links

between the layers.

If we were not to do this, our algorithm may become unsound. That is, a system may be assessed to be safe (i.e., unwanted regions of the output may be shown to be unreachable), but this could be the result of by approximations (rounding or truncation) due to the underlying floating point arithmetic.

To address this issue we use we use "epsilon" terms when encoding the network. These are terms used to link the layers of the network to allow for small perturbations when invoking the linear program solver. In combination with this, we change the objective function to minimise the sum of these epsilon terms, instead of simply using the constant 0.

Formally, for each layer, the constraint set changes as follows:

$$\begin{aligned} C_i = \{ & x_j^{(i)} \geq W_j^{(i)} x^{(i-1)} + b_j^{(i)} - \epsilon_j^{(i)}, \\ & x_j^{(i)} \leq W_j^{(i)} x^{(i-1)} + b_j^{(i)} + M\delta_j^{(i)} + \epsilon_j^{(i)}, \\ & x_j^{(i)} \geq 0, \\ & x_j^{(i)} \leq M(1 - \delta_j^{(i)}) \mid j = 1...|L^{(i)}| \} \end{aligned}$$

where ϵ_j are non-negative variables. Correspondingly, when encoding a neural network as linear program, we change the objective function to be $z = \sum_{i=2}^{k} \sum_{j=1}^{|L^{(i)}|} \epsilon_j^{(i)}$, which we then aim to minimise.

This entails that we aim to find an exact solution if possible but, if one exists with a small epsilon sum, we can still accept it if it is within the tolerance of the underlying floating point arithmetic. We do this by adding a further constraint of the form $\sum_{i=2}^{k} \sum_{j=1}^{|L^{(i)}|} \epsilon_j^{(i)} \leq t$, where t is the tolerance term.

In practice, for current computers we can take this to be $1e^{-6}$ which is one order of magnitude larger than the machine epsilon for 32-bit floating point numbers. We adopted this value in our experiments. However, when binary inputs are used, a larger tolerance value may be required because of the techniques used by solvers. We adopted a value of $1e^{-4}$ for these problems.

Finally, we present the full reachability verification algorithm; this is given by Algorithm 3.1.

It is worth taking the time to briefly explaining the key points in the algorithm. The most distinctive feature of the algorithm is how addition of constraints is broken up into two stages. The first stage involves simply adding variables and the second stage adding

Algorithm 3.1 Verifies reachability for a FFNN N with starting set I and target set O. Returns whether O is reachable from I through N

```
1: procedure VERIFY_REACHABILITY(N, I, O)
2:     Initialise new LP L
3:     for variables used in constraint set of I do
4:         x[1, j] = L.addVariable()
5:     end for
6:     for layer i = 2 to k do
7:         for j in layer i do
8:             epsilon[i, j] = L.addVariable(lower_bound=0)
9:             if layer i has a ReLU activation then
10:                delta[i, j] = L.addVariable(varible_type=BINARY)
11:                x[i, j] = L.addVariable(lower_bound=0)
12:            else
13:                x[i, j] = L.addVariable()
14:            end if
15:        end for
16:    end for
17:    for constraint C in constraint set of I do
18:        L.addConstraint(C)
19:    end for
20:    for layer i = 2 to k do
21:        W = weights(N, i)
22:        b = bias(N, i)
23:        for j in layer i do
24:            y = W[j] * x[i - 1] + b[j]
25:            L.addConstraint(x[i, j] ≥ y - epsilon[i, j])
26:            L.addConstraint(x[i, j] ≤ y + M * delta[i, j] + epsilon[i,
    j])
27:            if layer i has a ReLU activation then
28:                L.addConstraint(x[i, j] ≥ M * (1 - delta[i, j])
29:            end if
30:        end for
31:    end for
32:    for constraint C in constraint set of O do
33:        L.addConstraint(C)
34:    end for
35:    L.addConstraint(∑_{i,j} epsilon[i, j] ≤ tolerance)
36:    L.objective = (∑_{i,j} epsilon[i, j], MINIMIZE)
37:    return Has_Solution(L)
38: end procedure
```

the actual constraints.

The reason we adopt this two-phase approach is due to Gurobi, the program we use to solve the linear programs in the our implementation of this algorithm. Gurobi's API requires that we add variables explicitly before using them in constraints. To keep our algorithm as close to the implementation, we also use this structure in the pseudo-code algorithm as well.

We observe that the middle section of the algorithm closely reflects the set of constraints from the encoding of neural networks. We note that we incorporate our suggested changes from the section on floating point above. By doing this, we are able to retain soundness as discussed.

Finally, we see that we add a constraint in for the tolerance of the objective. The closer this value is to zero, the closer the solution to the linear program will be to solving the reachability problem exactly on the neural network. However, setting this value too low may mean that valid solutions are missed due to floating point. Thus, this value should tuned to a value depending on the exact problem being solved.

We note that this algorithm represents a sound and complete method for verifying reachability (modulo the tolerance term chosen) as the algorithm is able to definitively say whether or not the output is reachable from the input.

We formalise this in in the theorem below.

Theorem 3.3 (Soundness and completeness of reachability algorithm)**.** *Let N be a FFNN using only identity or ReLU layers.*

Let further $I \subseteq \mathbb{R}^m$ (respectively, $O \subseteq \mathbb{R}^n$) be a linearly definable set encoding the starting states (target states, respectively) where $m = |L^{(1)}|$ and $n = |L^{(k)}|$.

We have that the following statement holds: O is reachbable from I through N if and only if `VERIFY_REACHABILITY`(N, I, O) *returns* `true`*.*

3.5 Implementation and experiments

While we believe the theoretical contributions above are already significant, we think that any algorithm could not be properly evaluated without an actual implementation on which experiments could be performed. Such an implementation allow us to draw important conclusions in areas such as efficiency, scalability and applicability.

In this section, we aim to obtain information about these metrics. To do this, we start by describing how our implementation of our algorithm works before presenting a variety of case studies where we have successfully solved reachability problems. These studies are quite diverse and incorporate both control problems where research is very actively carried out to improve performance as well as traditional applications like classification problems.

For each of these case-studies, we briefly describe the problem followed by discussing the reachability specifications we verify. We present the results for each of these and finish by discussing the implications of these results. We will finish by summarising our findings and concluding our observations about the algorithm as a whole.

3.5.1 Implementation

The implementation of the reachability algorithm was more complex than it may first appear. This is because, as well as correctness of the algorithm, we were aiming for a well-engineered system so that this same code could be used in future research and tools.

Because of this, we had to be careful when deciding how to structure this implementation. As we want to reuse as much code as possible (especially because we wished to verify a large number of reachability specifications), the ideal structure would be exactly as with the pseudo-code where a single function would take a neural network and the starting and target sets.

However, as expected with any implementation of a pseudo-code algorithm, structuring the system like this was not feasible as each different case-study required a small amount of problem specific code when adding constraints for their starting sets. Moreover, a single line of pseudo-code can often become significantly more complex in the actual programming language.

Thus, by carefully considering the trade-off, the structure we decided on was to have a single class which acted as a "library" of sorts. In fact, this class actually forms part of the ReVerify library that we describe further in Chapter 7.

This class can be instantiated and used by scripts with one script being used for each reachability problem we wish to solve. By structuring the system this way, we have a large amount of code reuse but we also keep the system flexible to work with the wide variety of problems apply the algorithm to.

As we discussed in Chapter 2, all our implementation was carried out in Python with

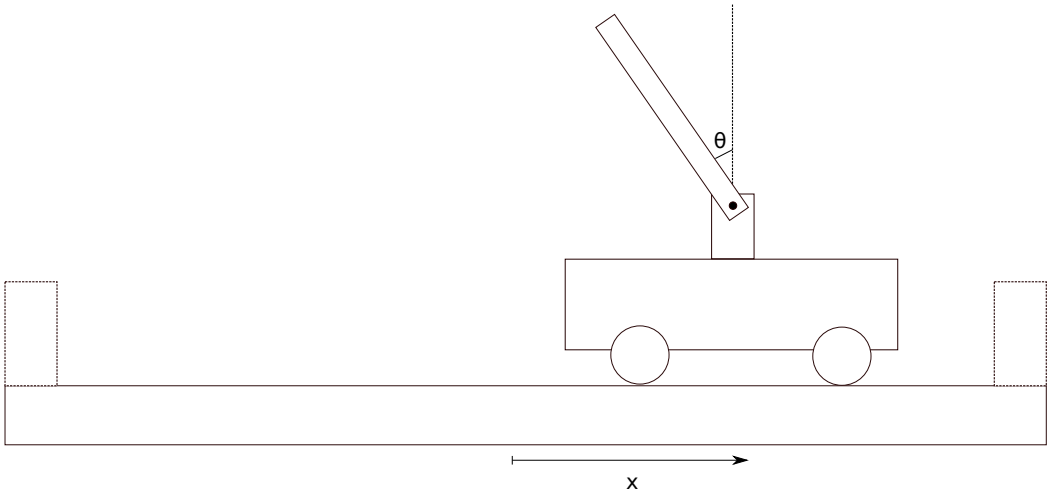


Figure 3.1: The Inverted pendulum on a cart physical system. The x axis denotes the displacement of the centre of mass of the cart from the centre of the track. θ is the angle between the pole and the vertical in a counter-clockwise fashion (i.e., the angle is positive when it is counter-clockwise from the vertical).

Keras being the neural network library of choice and Gurobi being used to solve all generated linear programs.

We note briefly that all the subsequent experiments in this section were run on an Intel Core i7-4790 CPU (3.600GHz, 8 cores) running Linux kernel 4.4 upon which we invoked Gurobi version 7.0.

3.5.2 Inverted pendulum on cart

The inverted pendulum on cart is a classic control problem which has long been used as a test-bed for new innovations in controller neural networks [1].

The physical system is composed by a cart moving along a frictionless track with bounds at either ends of the track. Attached to the centre of the cart through the use of a frictionless and unactuated joint is a pole; the pole acts as an inverted pendulum. A diagram of the system is given in Figure 3.1.

In control terms, the problem can be expressed by using two state variables and their derivatives [1].

- Position of the cart on the track, denoted by x and bounded by ± 2.4.
- Speed of the cart, denoted by $\dot{x}$ and unbounded.
- Angle of the pendulum (counter-clockwise), denoted by θ and bounded by $\pm 15°$.
- Angular velocity of the pendulum (counter-clockwise), denoted by $\dot{\theta}$ and unbounded.

The output of the controller (i.e. the network) at every discrete time step is a signal for the application of a force of $+10N$ or $-10N$ (where the direction is aligned with x). Intuitively, the aim of the controller is to balance the pendulum on the cart for as long as possible while, at the same time, remaining both in the bounds of the track and in the bounds of the angle of the pendulum.

To train the network, we use the Double Deep Q-Learning algorithm, a reinforcement learning algorithm. After training the resulting network obtained can be described as follows:

- The input layer consists of 4 input nodes, one for each of the variables of the system.
- The subsequent three hidden layers consist of 16 nodes; each layer uses the ReLU activation function.
- The output layer consists of 2 nodes denoting the "q-value" of moving left and right respectively. The higher the q-value, the higher is the predicted reward for the action. The output layer does not use a ReLU function as is standard for most networks.

In view of the encoding discussed above we can now proceed to verify the behaviour of the neural-network trained for the IPCP. We consider the following specifications where in each case S is a tuple of form $(x, \dot{x}, \theta, \dot{\theta})$.

1. Is it ever the case that $Q(S, 10) \not\geq Q(S, -10) + 100$ where $S = (0, 0, -5, 0)$? Intuitively, this says that force of $10N$ has a Q-reward of at least 100 units greater than $-10N$ for the given state S. This expresses the fact the controller attempts (in the strongest possible sense within the bounds of the problem) to move the cart to the right when the pendulum is leaning to the right and all other factors are unimportant.
2. Is it ever the case that $Q(S, 10) \not\geq Q(S, -10) + 100$ where $S \in \{(x, \dot{x}, \theta, \dot{\theta}) \mid |x| \leq 0.5, |\dot{x}| \leq 0.2, -5 \leq \theta \leq -4, |\dot{\theta}| \leq 0.1\}$? This is the same specification as above but

to be checked on a larger states of configurations.

3. Is it ever the case that $Q(S, 10) \not\geq Q(S, -10) + 10$ where $S \in \{(x, \dot{x}, \theta, \dot{\theta}) \mid x \leq -2, |\dot{x}| \leq 0.2, -2 \leq \theta \leq -1.5, |\dot{\theta}| \leq 0.25\}$? This represents the fact that the controller attempts to move the cart to the right when the pendulum is not at risk from falling over but the cart is almost out of left hand side track bound.

4. Is it ever the case that $Q(S, 10) \not\geq Q(S, -10) + 10$ where $S \in \{(x, \dot{x}, \theta, \dot{\theta}) \mid x \leq -2, |\dot{x}| \leq 0.4, -2 \leq \theta \leq 1, 0 \leq \dot{\theta} \leq 0.1\}$? This is a relaxation of the above specification to analyse a larger set of configurations.

By solving these problems, we obtain the following results.

1. No solution could be found satisfying the constraints on the input and output of the network. We conclude that our synthesised controller does strongly prefer to apply $10N$ to balance the pendulum in those circumstances.

2. As above no solution could be found. Again, we conclude that in all the region explored the behaviour of the synthesised controller is correct.

3. Again, no solution could be found satisfying both the inputs and the outputs. This indicates that the controller attempts to return the cart to the centre of the track when it is one side of the track within the range of parameters above.

4. The solver reported the solution $x = (-2.0, -0.4, -0.15, 0.1)$ (approximation shown) for the problem above. This shows that the controller applies the force in what is, intuitively, the incorrect direction when the configuration of the system is as above. Note also that in this situation the angle of the pendulum would also suggest an application of the force in the positive direction.

The analysis conducted above shows that the neural network does not satisfy our specifications as put forward. The values found by the solver can be fed to the neural network to confirm the result. We have effectively found a "bug" in the neural network by using a formal encoding into an LP problem.

Table 3.1 gives the time taken for solving the reachability problems described above. We see that in all cases, the time taken to solve the problem was less than 1 sec. To ensure that this was not simply because of the chosen properties we also checked several other reachability specifications (which were not included) to evaluate the performance degradation as a function of the specification. We could not find specifications that could not be solved in under 1 sec. We conclude that the methodology presented can be used to evaluate any reachability problem for the given network.

Problem	Vars (Continuous, Binary)	Time (s)
Inv. Pen. 1	108, 31	≤ 0.01
Inv. Pen. 2	140, 39	0.02
Inv. Pen. 3	142, 41	0.03
Inv. Pen. 4	143, 42	0.04

Table 3.1: Experimental results for cartpole problem. Vars column refers to the number of variables in the LP: both continuous and binary.

Figure 3.2: A diagram of the inverted pendulum problem.

We stress the importance of this result as comparable techniques, e.g. testing, are in incomplete and may take considerably longer to identify the need for further training.

3.5.3 Balancing a pendulum (inverted pendulum)

The pendulum balance (aka the inverted pendulum) problem involves attempting to keep a pendulum balanced upright. Obviously this system is highly unstable so the agent is required to apply a bounded torque at the pivot point. A diagram of the physical system can be seen in the Figure 3.2.

In control terms, the problem can be expressed by using one state variable and its derivative.

- Angle of the pendulum (counter-clockwise from the vertical line), denoted by θ, bounded by $[0, 2\pi]$.

- Angular velocity of the pendulum (counter-clockwise), denoted by $\dot{\theta}$ and bounded by $\pm 8°$.

The algorithm used to train the network in this problem is called Deep Deterministic Policy Gradient, another reinforcement learning algorithm. After training, the resulting network obtained can be described as follows:

- The input layer consists of 3 input nodes. Instead of using the angle directly as input, the cosine and sine are used, to make training more efficient. Thus, the input to the network is the vector $[\cos(\theta), \sin(\theta), \dot{\theta}]$.
- The subsequent three hidden layers consist of 32 nodes; each layer uses the ReLU activation function.
- The output layer consists of 1 node which denotes the torque produced by the agent.

As the network relies on non-linear trigonometric functions; we generate a piece-wise linear approximation for these. The details of this encoding are omitted but can be found in our implementation.

We solve four reachability specifications. The first three verify properties which are desirable for the network to have. The fourth does not have such a meaning and is more intended as a test for the algorithm to force the whole of the state space to be searched for a particular value of the output.

The specifications and the corresponding results that our algorithm found are as follows

- **Are only small positive torques applied when close to vertical?** No solution could be found so we conclude the controller only applies small positive torques in this configuration.
- **Are only small negative torques applied when close to vertical?** A solution was found for the state S = (0.086, 0.996, 0.086, 0.001) (to 3 d.p.) which indicates that the network does not perform perfectly in this configuration.
- **Is a positive torque applied when close to vertical downwards and a positive velocity is present?** No solution could be found so we conclude the controller only applies positive torques in this configuration.
- **Does there exist any state in the input space where a torque of 1 is applied?** A solution was found for the state S = (0.81, 0.55, -1.84) (to 2 d.p.).

Problem	Vars (Continuous, Binary)	Time (s)
Pendulum 1	154, 41	0.05
Pendulum 2	154, 41	0.02
Pendulum 3	154, 41	0.02
Pendulum 4	161, 48	0.65

Table 3.2: Experimental results for inverted pendulum problem. Vars column refers to the number of variables in the LP: both continuous and binary.

Table 3.2 gives the time taken for solving the reachability problems described above. We see that in all cases, the time take to solve the problem was again less than 1 sec but slightly higher than for the cartpole problem. This corresponds with the slight increase in the number of continuous variables but no increase in the number of binary variables. However, the final specification which took significantly longer than that others; we believe this to be because of the increase in binary variables for this problem.

3.5.4 Swing up acrobot

An acrobot is similar to a pendulum but instead of a single unbroken rod, the rod is broken into two pieces which are attached by a frictionless joint. At the joint, a fixed torque can be applied either clockwise or anti-clockwise or no torque applied at all. The aim of the agent is to get the the end of the lower link above a certain point by applying the torque. A diagram of the physical system can be seen in the Figure 3.3.

In control terms, the problem can be expressed by using two state variables and their derivatives.

- Angle of the first joint (counter-clockwise from the vertical line downward), denoted by θ_1, bounded by $[0, 2\pi]$.
- Angle of the second joint (counter-clockwise from the vertical line downward), denoted by θ_2, bounded by $[0, 2\pi]$.
- Angular velocity of the first joint (counter-clockwise), denoted by $\dot{\theta_1}$, bounded by $\pm 4\pi$.
- Angular velocity of the second joint (counter-clockwise), denoted by $\dot{\theta_2}$, bounded by $\pm 9\pi$.

The algorithm used to train the network in this problem is called Deep Deterministic

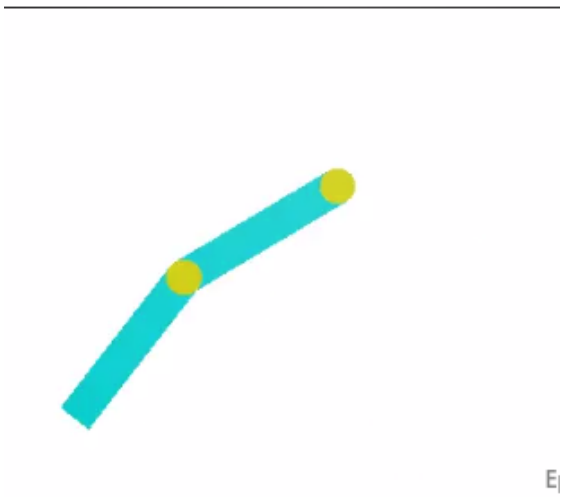


Figure 3.3: A diagram of the acrobot problem.

Policy Gradient, another reinforcement learning algorithm. After training the resulting network obtained can be described as follows:

- The input layer consists of 6 input nodes. Instead of using the angles directly as input, the cosine and sine are used, to make training more efficient. Thus the input to the network is the vector $[\cos(\theta_1), \sin\theta_1, \cos\theta_2, \sin\theta_2, \dot{\theta_1}, \dot{\theta_2}]$.
- The subsequent two hidden layers consist of 64 and 168 nodes respectively; each layer uses the ReLU activation function.
- The output layer consists of 3 nodes denoting the "q-value" of applying a torque anti-clockwise, doing noting and applying a torque clockwise respectively. The higher the q-value, the higher is the predicted reward for the action. The output layer does not use a ReLU function as is standard for most networks.

As this network again relies on non-linear trigonometric functions; we generate a piece-wise linear approximation for these.

As with the inverted pendulum problem, the first two specifications verify desirable properties while the second two force the solver to search large parts of the state space. Compared to the other problems, the first two specifications test very complex properties which require multiple invocations of the LP solver.

The specifications and the corresponding results that our algorithm found are as follows

- **When only a small clockwise rotation is required, is it the case that clockwise torque is preferred to no torque which is preferred to an anti-clockwise torque (i.e. Q(clockwise) $\geq$ Q(no torque) and Q(no torque) $\geq$ Q(anti-clockwise) and Q(clockwise) $\geq$ Q(anti-clockwise))?** This problem required invocation of the LP solver three times - once for each of the three inequalities. No solution could be found for the first inequality but solutions could be found for the second and third. This indicates that the network is partially correct but does not perform perfectly in this configuration.

- **When only a small anti-clockwise rotation is required, is it the case that anti-clockwise torque is preferred to no torque which is preferred to an clockwise torque (i.e. Q(anti-clockwise) $\geq$ Q(no torque) and Q(no torque) $\geq$ Q(clockwise) and Q(anti-clockwise) $\geq$ Q(clockwise))?** This problem required invocation of the LP solver three times - once for each of the three inequalities. No solution could be found for any of the three inequalities. This indicates that the network is behaves exactly as expected in this configuration.

- **In a large part of the state space, does there exist a state where Q(none) $\geq$ Q(right) + 1?** No solution could be found.

- **In a very large part of the state space, does there exist a state where Q(left) $\geq$ Q(right) + 4?** No solution could be found.

Acrobot 1	579, 162	0.48 (3 problems)
Acrobot 2	569, 152	0.66 (3 problems)
Acrobot 3	590, 173	3.31
Acrobot 4	609, 192	47.28

Table 3.3: Experimental results for acrobot problem. Vars column refers to the number of variables in the LP: both continuous and binary.

Table 3.3 gives the time taken for solving the reachability problems described above. We see that the performance characteristics of acrobot are vastly different to the other two problems to date with one specification taking more than 45 seconds to solve.

However, we make the argument that this is still good performance. The specifications verified here are significantly more demanding than the other experiments. This is due both to the complex and high dimensional state space as well as the relatively large number of neurons in each layer. The third and fourth specifications for acrobot are especially demanding as they require a full search of a large part of a relatively

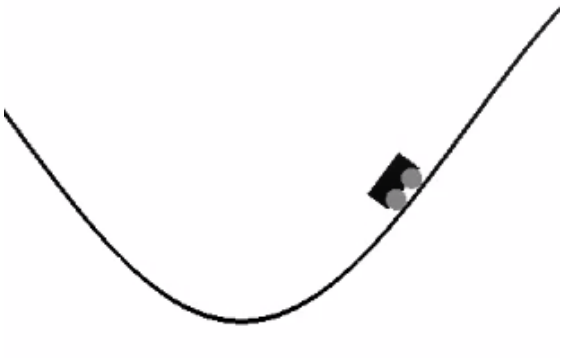


Figure 3.4: A diagram of the mountain car problem.

high dimension state space to find a solution. Taking this into account, we believe the performance to be more than acceptable.

We note that there does appear to be a relationship between an increase in the number of binary variables and the time taken to solve the algorithm. This was a trend we also observed in the inverted pendulum problem.

3.5.5 Scaling a mountain using an underpowered car

The mountain car problem details a car which is attempting to scale a hill to the right. The engine can apply a fixed force left or right or apply no force. However, the engine is not powerful enough to scale the hill. Instead, the car must first scale a smaller hill to the left to build up momentum. A diagram of the physical system can be seen in the Figure 3.4.

In control terms, the problem can be expressed by using one state variable and its derivative.

- Position of the cart on the track, denoted by x and bounded by $[-1.2, 0.6]$.
- Speed of the cart, denoted by $\dot{x}$ and bounded by ± 0.07.

To train the network, we again use the Double Deep Q-Learning algorithm, a reinforcement learning algorithm. After training the resulting network obtained can be described as follows:

- The input layer consists of 2 input nodes, one for each of the variables of the

system.

- The subsequent two hidden layers consist of 50 and 190 nodes; each layer uses the ReLU activation function.

- The output layer consists of 3 nodes denoting the "q-value" of moving left, doing nothing and moving right respectively. The higher the q-value, the higher is the predicted reward for the action. The output layer does not use a ReLU function as is standard for most networks.

We solve four reachability specifications. The first three verify properties which are desirable for the network to have. The fourth does not have such a meaning and is more intended as a test for the algorithm to check the whole state space for a certain property.

The specifications and the corresponding results that our algorithm found are as follows

1. **When close to the goal (on the right hill) with enough momentum, does the network favour moving to the goal over moving down the hill?** No solution could be found so we conclude the controller does apply a force to the right in this configuration.

2. **When on the smaller hill to the left with some velocity to the left, does the network try to build further momentum by applying a force to the left?** No solution could be found so we conclude the controller does apply a force to the left in this configuration.

3. **When on the right hill without any velocity to the right, does the network try to build momentum by applying a force to the left to move down the hill?** A solution was found for the state S = (0.3, -0.01) (to 2 d.p.) which indicates that the network does not perform perfectly in this configuration.

4. **Does there exist any state in the input space where no force is preferred over applying a force to the left?** A solution was found for the state S = (-1.2, 0.03) (to 2 d.p.).

Table 3.4 gives the time taken for solving the reachability problems described above. We see that in all cases, the time take to solve the problem was again less than 1 sec but slightly higher than for the cartpole problem.

We see that the continuous and binary variables have both increased but it does not seem to have affected the running time of the solver. This is rather surprising and does not correspond well with the other results which point to a significant effect of

Problem	Vars (Continuous, Binary)	Time (s)
Mountain Car 1	406, 117	0.06
Mountain Car 2	404, 115	0.04
Mountain Car 3	404, 117	0.02
Mountain Car 4	407, 118	0.04

Table 3.4: Experimental results for mountain car problem. Vars column refers to the number of variables in the LP: both continuous and binary.

the number of binary variables on the time taken. One explanation may be that the low input dimensionality (2 dimensions) means the solver is not being forced to search hard for solutions for the problems. It may be that we needed to use more complex specifications.

3.5.6 Reuters text classification

This neural network is intended to solve the problem of classifying articles from their content [51]. The network is composed of a binary input layer followed by a hidden layer with a ReLU activation function followed by the output layer. The structure of network is thus rather shallow, but contains a large number of input and hidden neurons. It was trained using supervised learning.

To evaluate the performance of the approach, we fixed all but 50 of the inputs to known values and attempted to carry out reachability analysis on the remaining inputs. We also neglected the use of the softmax function to reduce the complexity resulting from its use.

We were able to replicate the existence of an input for a (pre-softmax) output of the network for which we knew that a binary input existed. We were able to solve the LP problem and find the corresponding values in just under 45 secs. As above, considering the size of the hidden layers and the number of binary variables present in the corresponding problem, we find the performance to be attractive.

3.5.7 MNIST Image Recognition

This neural network was put forward to perform image recognition on the MNSIT handwritten numeric digits dataset [63]. The network is composed of a convolutional part with several filters and a max-pooling layer followed by a hidden feed-forward layer and

an output layer. It was trained using supervised learning.

We only considered the hidden and output layers of the network as these are the feed-forward portion of the network. As in the previous example the size of these layers is between 10^2 and 10^3 nodes. As in the previous experiment we attempted to find an input for some output for which we know an input exists. As before, we did not use softmax function when encoding the output.

Problem	Layer Sizes	Vars (Continuous, Binary)	Time (s)
Reuters	1000, 512, 46	1546, 526	40.30
MNIST	4608, 128, 10	5002, 128	8.54

Table 3.5: Experimental results for Reuters and MNIST problems. Vars column refers to the number of variables in the LP: both continuous and binary.

Our algorithm was able to find the input in just over 8 secs. Given the size of the network and the number of variables in the corresponding problem, we again evaluate the performance positively. Comparing to the Reuters, and referring to Table 3.5, we conjecture that the large increase of binary variables in the problem creates the large performance gap between the Reuters dataset and MNIST.

In summary, the results suggest that the methodology developed, when paired with the optimisations here studied, can solve the reachability problem for several neural networks of interest. In particular we were able to solve reachability analysis for deep nets of 3 layers of significant size. The experiments demonstrate that performance depends on a number of factors the most important being the size of the state space searched, the number of variables (especially binary variables), and the number of constraints.

In this chapter, we gave a novel definition for the reachability property for ReLU feed-forward neural networks. We showed how such networks could be encoded as a set of linear constraints. We then gave a sound and complete algorithm to verify reachability on these networks, taking into account floating point errors. Finally, we evaluated the algorithm on a wide range of problems and demonstrated that the algorithm was performant across of all of them.

4 Encoding functions with linear constraints using neural networks

In this chapter, we discuss how we can combine multiple neural networks to accurately approximate functions which we can then encode as a set of linear constraints. Our motivation for this is due to techniques in upcoming chapters, where we rely on being able to model functions such as sine and cosine in a linear program. These functions are obviously non-linear so cannot be directly represented using linear functions.

We will discuss why techniques such as linearisation either on the domain as a whole or piecewise are not sufficient for our purposes. We then move on to talk about why neural networks are a good solution to this problem. We finish with a case-study considering this technique may work for the inverted pendulum problem.

This chapter does not contribute directly towards the achieving the aims of the project. However, the techniques we describe are absolutely crucial to our work on verification of LTL in both approximating non-linear functions used in defining the environment function as well as pre- and post-processing the input and output respectively of agent neural networks.

Moreover, while the technique we describe here of approximating functions using neural networks is not new, we believe that the idea of approximating such functions to generate linear approximations is in-fact novel. Moreover, we note that this technique is **not** limited to the context of network systems or LTL.

In fact, we believe this is essentially a stand-alone technique which could be applicable to any field where one requires an accurate linear approximation of an arbitrary non-linear function. It is merely coincidental that LTL verification is one area where it is very useful.

4.1 Alternatives techniques

In this section, we discuss why we could not simply use complex functions directly and some alternative ways of linearising functions and why they fail to meet our needs.

4.1.1 Directly utilising the complex functions

An alternative to any approximation technique, is to simply directly use the complex functions themselves. These functions will usually define the physics of some problem and so will usually be defined by a small system of equations which appear to be reasonably simple. However, while mathematically they are easy to understand and can be written down compactly, for verification they pose great challenges.

For example, for the inverted pendulum cart-pole problem presented in the previous chapter, the dynamics of the system are modelled by just three equations. However, these equations contain several sine, cosine and quadratic terms; some of these quadratic terms are quadratic in the cosine of the angle!

Merely introducing quadratic equations vastly increases the complexity of the problem. Gurobi can handle some highly specialised forms of quadratic equations but it is not a general purpose quadratic solver by any means. Even if we were able to model quadratic equations, doing the same with transcendental functions like sine and cosine and taking quadratic terms of those is beyond the capability of any linear or quadratic solver.

Even in the field of SMT and general purpose optimization, these functions are extremely expensive to model and can create undecidable problems. This is clearly not ideal as we would need to explore a whole new type of optimization tool with no guarantees as to whether the problem is even feasible. Moreover, we lose all the work we have done to date modelling neural networks as linear programs and the related optimizations and analyses.

As it is important that our technique be reasonably efficient at least for small systems and requiring decidability, and wishing to reuse our work to date on linear programming, we see that this method does not fit our needs.

4.1.2 Linearising equations

Simplifying complex equations into linear ones is a common technique in engineering and physics; this is known as "linearisation". The basic idea behind the technique is to pick a point around which a Taylor expansion is performed and use up to the linear term in the expansion.

For example for the function $\exp(x)$, we can take the Taylor expansion around the point $x = 0$ to give the linear approximation

$$\exp(x) = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \approx 1 + x$$

As is obvious, while the expansion leads to an accurate approximation close to the chosen point, as one moves away from this point, the expansion becomes increasingly inaccurate. Far away from the point, it becomes completely non-representative of the true function. This problem is especially bad with a function like $\cos(x) \approx 1$ which is only remotely accurate around $x = 0$.

For our purposes, it is important that such approximations are accurate as verification would quickly become unsound if there were large discrepancies between our model of the environment and the true equations modelling it. For this reason, we cannot utilise this method to model non-linear functions.

4.1.3 Piecewise-linear approximations

Another technique which can be used to turn a non-linear function into some combination of linear ones is known as piecewise linearisation. Instead of trying to represent the function on the whole domain using a single linear function, piecewise linearisation breaks up the domain into intervals and uses a linear approximation in each interval.

This vastly improves the accuracy compared to simply linearising the equation across the whole domain. Each linear approximation is only valid for a small part of the domain but over this part it is a very good approximation. Moreover, the number of intervals is easy to modify so the accuracy of the approximation can be easily tweaked to any level necessary.

Implementing this technique in a linear program involves, for each non linear operation, creating n binary variables: one for every such piecewise interval. A type 2 SOS constraint is then added over these n variables to ensure that only one interval is picked to

provide the approximation.

However, the major disadvantage with this method is the vast increase in binary variables and the use of SOS constraints. For example, pendulum cart-pole performs 6 non-linear operations. To generate a sensible approximation for each of these operations takes at least 20 intervals, which leads to a total of 120 extra binary variables. Moreover, it also adds 6 SOS constraints each acting over 20 variables.

As discussed in the previous chapter, for the type of LP we create, SOS constraints are usually expensive. Moreover, if one non-linear function depends on the output of another (e.g. $\cos^2\theta$ depends on $\cos\theta$) then then these constraints may become even more expensive.

While this technique might be feasible, we did not investigate it deeply due to these predicted issues. Future research into this technique is required to fully decide if it is appropriate for the technique we present in the next chapter.

4.2 Simple non-linear functions

Having discussed these alternative techniques we now consider why neural networks in particular are appropriate for this function.

A fundamental principle of neural networks is that they can approximate complex functions to arbitrary degree of accuracy; this is known as the universal approximation theorem. Our work in reachability has already demonstrated how we can then turn this network into a set of linear constraints.

Thus, the idea is to show how we can indirectly approximate a non-linear function using a set of linear constraints by first training a neural network to approximate the function and then utilising our definitions and theorems from Chapter 3 to turn this neural network into a set of linear constraints.

Much of the work for this has already been done in previous chapters but in this section we will consolidate this into a few definitions and a theorem. We will utilise these in upcoming chapters as part of our LTL verification algorithm.

We start by defining the encoding of the approximation for functions which can be represented directly by a neural network.

Definition 4.1. Let $g : \mathbb{R}^m \to \mathbb{R}^n$ be a non-linear function with N a FFNN with each layer having a ReLU or linear activation function such that its computed function $f : \mathbb{R}^m \to \mathbb{R}^n$ approximates g (informally that the difference between the output of the two functions is "small").

Then, the set of *neural network linear constraints* approximating the function g are precisely the equal to the set of linear constraints encoding the network N.

4.3 Linear compositions of non-linear functions

While it is the case that any function can be represented with arbitrary accuracy using a neural network, training such a network on the other hand is a very hard problem. In fact for more complex functions, this can actually end up being essentially impossible.

For our use-case, we often have rather complex functions which cannot be easily represented using a single neural network. As an example consider the function

$$f(x) = (\sin x, \cos x)]$$

For this function, calculating the sine and cosine separately using two networks and composing the would be much more accurate rather than training a single neural network to perform both simultaneously.

As a more complex example, consider the function

$$f(x) = \sin x + \frac{10}{13} \cos x$$

Again, it would be better to compute the sine and cosine using two networks and compute the linear combination of the two directly in the linear program itself. In essence, we seek to isolate the non-linear sections to simple functions, approximate these and put them together directly using a piecewise linear function.

This is precisely what we seek to formalise in the following definition.

Definition 4.2 (Linear constraints for composed functions)**.** Let $g^i : \mathbb{R}^{m_i} \to \mathbb{R}^{n_i}$ be non-linear functions with N^i the corresponding FFNN such that each layer of each network has a ReLU or linear activation function with each computed function $f^i : \mathbb{R}^{m_i} \to \mathbb{R}^{n_i}$ approximating g^i well for $1 \leq i \leq k$. Further let $c : \mathbb{R}^{n_1} \times ... \times \mathbb{R}^{n_k} \to \mathbb{R}^c$ be a piecewise linear function.

Define the function $o : \mathbb{R}^{m_1} \times ... \times \mathbb{R}^{m_k} \to \mathbb{R}^c$ as $o(x^1, ..., x^k) = c(f^1(x^1), ..., f^k(x^k))$

Let z be a vector of fresh LP variables of dimension c which would denote the output of the o function. Let C^i be the set of linear constraints encoding each network N^i with the vector of LP variables (of dimension n_i) for the last layer of each network denoted b^i.

Then, the set of *neural network linear constraints* encoding the composed function o is defined to be

$$C_o = \cup_{i=1}^{k} C^i \cup \{z = c(b^1, ..., b^k)\}$$

With this definition complete, we briefly state a theorem which ties the satisfaction of thesis set of constraints to precisely approximating the function well. We do not prove this theorem as it is rather tedious (in the style of those proved in the previous chapter) but it should be intuitive to grasp the idea of how such a proof might appear.

Theorem 4.1. *Let o be a composed function and C_o be the set of neural network linear constraints encoding the composed function.*

Let the vector of LP variables for the first layer of each network N^i associated to o be denoted a^i. Let z be a vector of LP denote the output of the o.

Then we have that the following: for any solution for the variables above labelled $\mathbf{a}^i$ and $\mathbf{z}$, they satisfy the constraints if and only if $o(\mathbf{a}^1, ..., \mathbf{a}^k) = \mathbf{z}$.

Informally, we state that the set of constraints is satisfied if and only if this solution also provides the arguments and output of the function the constraints are attempting to capture. This essentially states that our constraint encoding captures exactly the meaning of the function o.

In this chapter, we have demonstrated how for any linear combination of non-linear functions, we can first train FFNNs to approximate the non-linear portions and then define a set of linear constraints which encodes each FFNN followed by a single constraint which encodes the linear combination itself. We will make use of this idea quite extensively in temporal verification so while simple, the ideas in this chapter are vital for understanding upcoming chapters.

5 Network systems

Up until this point we have been considering a single neural network which responds to input to produce an output or "action" and performed verification of reachability specifications on these networks. However, these networks are only half the story.

We have been treating the environment these networks operate in as unrelated to our verification work. However, agents and environments do not operate in isolation: the "action" produced by the network modifies the environment and the state of the environment directly affects the action chosen by the network.

First, we define the concept of agent networks. This structure formalises the concepts we made use of in the chapter on reachability. Then, building on the work we carried out in the previous chapter, we discuss how we can represent the physics underlying control problems as environment functions and turn them into a set of constraints.

We the consider the composed structure known as a network system and consider the encoding of this system using linear constraints. We finish by carrying a case study with the inverted pendulum system.

5.1 Agent networks and environment functions

In this section we formalise some of the language we will use throughout the remainder of the project. Specifically, we will define agent networks and environment functions in turn and give an intuition of the definitions on what these terms mean as well.

Definition 5.1 (Agent Network)**.** Let N be an FFNN, S be any set representing the possible states the "environment" might be in and A be any set representing the "actions" that are available to an agent.

An *agent network* is a tuple (N, S, A, s, a) where

- s is a function $s : S \to R^m$ representing a transformation of the current state to

a tuple which can be used as input for N

- a is a function $a : R^n \to A$ which converts the output of N into the action taken by this agent.

with $m = |L^{(1)}|$ $n = |L^{(k)}|$ and k the number of layers in N.

The sort of neural networks we intend this definition to apply to are precisely the networks we have been studying until this point in the project. For example, all the networks (with the exception of the Reuters and MNIST networks) in Chapter 3 fall in this category. These networks solve "control problems". Informally, these are problems where the network is producing a result which is controlling some physical entity which is governed by the laws of physics.

We note that the set of states and actions is intentionally left vague to allow a problem specific definition. For example, in the inverted cart-pole problem, the set of states would be $S = \{(x, \dot{x}, \theta, \dot{\theta}) || x| \leq 2.4, |\theta| \leq \frac{12}{180} 2\pi\}$ and set of actions would be $A = \{\text{left}, \text{right}\}$ representing a force being applied to move to the cart to either the left of the right.

The function s would tie very closely to the exact structure of the network; in the simplest case it would simply be the identity function (i.e. $s(x, \dot{x}, \theta, \dot{\theta}) = (x, \dot{x}, \theta, \dot{\theta})$) but as we shall discuss later in this chapter, agents often perform better when there is pre-processing of inputs. Allowing for an arbitrary function s allows for flexibility in this regard.

The function $a : R^2 \to A$ would be defined

$$a(l, r) = \begin{cases} \text{left} & l \geq r \\ \text{right} & \text{otherwise} \end{cases}$$

The definition of the function reflects the fact that the output of the network is a prediction of the reward gained from each action; it is obvious that we would pick the action with the highest reward so we move right when the reward for it is higher and vice versa. The case where both outputs of the network are equal means that neither action is preferred so we have arbitrarily chosen the left action to be preferred in this case.

Definition 5.2 (Environment Function)**.** Let S be any set representing the state of the environment and A be any set representing the "actions" that are available to an agent in the environment.

An *environment function* is any function $e : S \times A \to S$.

Environment functions essentially take the current state of the network and the action produced by the agent and produce the new state of the network. This essentially allows them to represent the "physics" of the problem.

These can be equations such as the equations of motion or (as we shall see) the relation between torque on a pendulum and the rotational velocity. Again the vagueness of the states and actions is intentional; similar to agent networks, we wish to allow for problem specific definitions for both.

5.2 Network systems

With these definitions, we now move on to talk about how we can combine agent networks and environment functions together to form a complete system. We draw inspiration from the field of interpreted systems where agents and environments are similarly combined to form a system. First, however, we define an auxiliary concept which will help in defining network systems.

Definition 5.3 (Computed function of agent network)**.** Let $G = (N, S, A, s, a)$ be an agent network. Let f be the computed function of N.

We define the *computed function* of the agent network G $g : S \to A$ as

$$g(st) = a(f(s(st)))$$

The computed function is informally just a composition of the three functions inside the agent network. We convert the current state into a real vector which encodes the state into some the form required by the neural network. The neural network is used to compute another real vector which encodes the reward associated to each available action. This vector which is then converted into one of the actions in A.

With this definitions we can finally consider the composition of the agent network and environment function: we call this a network system.

Definition 5.4 (Network System)**.** Let $G = (N_{agent}, S, A, s, a)$ be an agent network (as in Definition 5.1) and $e : S \times A \to S$ be an environment function (as in Definition 5.2). Let g be the computed function of G.

Then we define a *network system* as the tuple $NS = (G, e, t)$ where $t : S \to S$ is a

transition function defined as

$$t(st) = e(st, g(st))$$

A network system is just a convenient abstraction representing the transition function from one point in the state space to another point in the state space. We see that the agent gets a copy of the current state of the environment which allows it to produce an action i.e. a reaction to the current state of the environment. This is then passed to the environment function which also gets a copy of the current state. The environment function then produces the new state of the network and the process can repeat ad infintium.

5.3 Encoding network systems using linear constraints

We now discuss how we can turn a whole network system into a set of linear constraints. In the subsequent chapter, we will begin discussing how we can verify LTL on network systems: it is fundamental to this work that we are able to query information about the output of the network (for example the maxima and minima of each dimension of the state space). To perform these queries, we need to solve linear programs and for that we will need the encoding we discuss here.

To begin, we briefly discuss how such an encoding may work. A network system is fundamentally composed of an agent network and an environment function. An agent network itself is composed of a core neural network and "glue" functions which transform from the common sets of S and A into specialised real vectors to pass to the neural network.

The first thing we realise when attempting to define an encoding is realise that, while we have made all our definitions agnostic to the exact structure of S and A, our encoding cannot be so. In fact, we must impose the requirement that $S \subseteq \mathbb{R}^m, A \subseteq \mathbb{R}^n$ for some m and n to allow any encoding to be defined.

The other important point of note is that our "glue" functions (s and a for action networks) and our environment function should either be linear functions or, more generally, can be represented as linear combinations of non-linear functions. Then utilising our work from the previous chapter, the non-linear functions can be approximated using neural networks which are themselves then encoded to a set of linear constraints.

Formalising this intuitive understanding, we build up a set of linear constraints encoding

a network system by first encoding the agent networks and environment functions before composing the two.

Definition 5.5 (Linear constraints encoding a agent network)**.** Let $G = (N, S, A, s, a)$ be an agent network where $S \subseteq \mathbb{R}^m, A \subseteq \mathbb{R}^n$ with s and a being linear combinations of non-linear functions.

Let C be the set of linear constraints encoding N, C_s be the set of neural network linear constraints encoding s with some set of neural networks N^s and C_a be the set of neural network linear constraints encoding a with some set of neural networks N^a.

Take c_{in} and c_{out} to be the vector of LP variables encoding the input and output of N respectively in C. Take s_{out} to be the vector of LP variables encoding the output for the encoding of s in C_s. Take a_{in} to be the vector of LP variables encoding the input for the encoding of a in C_s.

We define the set of linear constraints encoding G as follows:

$$C_g = C \cup C_s \cup C_a \cup \{s_{out} = c_{in}, c_{out} = a_{in}\}$$

We see that, intuitively, the encoding of an agent network involves encoding its neural network followed by adding constraints to encode the state and action function and finishing by connecting up each of the individual pieces. When "glue" these pieces together, we follow exactly how the computed function of agent networks is defined.

Using our work from the previous chapter, we see how we can encode environment functions.

Definition 5.6 (Linear constraints encoding a environment function)**.** Let $e : S \times A \to S$ be an environment function where $S \subseteq \mathbb{R}^m, A \subseteq \mathbb{R}^n$ with e being a piecewise linear composition of of non-linear functions.

The set of linear constraints encoding the environment function is simply the set of neural network linear constraints encoding the function itself for some set of neural network N^e encoding the non-linear functions.

We finish by considering the composition of the above by encoding a full network system.

Definition 5.7 (Linear constraints encoding a network system)**.** Let $NS = (G, e, t)$ be a network system. Let C_g be the set of linear constraints encoding the agent network G and C_e be the set of linear constraints encoding the environment function

e.

Let st be a vector of LP variables representing the current state of the environment and st_{new} be a vector of LP variables representing the new state of the environment. Let further g_{in} and g_{out} be vectors of LP variables used when encoding the input and output respectively of C_g. Finally, let e_{in} and e_{out} also be vectors of LP variables used when encoding the input and output respectively of C_e.

We define the set of linear constraints encoding NS as follows:

$$C_{ns} = C_g \cup C_e \cup \{g_{in} = st, e_{in} = (st, g_{out}), e_{out} = st_{new}\}$$

Again, perhaps more clearly than the agent network case, we see this encoding puts together the two encodings of the agent network and environment function and adds some constraints which glue the pieces together. The structure of the definition of the transition function is very visible in these extra constraints.

5.4 Case study: Inverted Pendulum

To demonstrate how the structures we have defined in this chapter are useful in practice, we now consider a case study. Specifically, we will show how the inverted pendulum control problem which we described in Chapter 3 can be defined in terms of network systems. First, we start by deciding the state and actions spaces with which we wish to work.

From the description of the problem, we see that the two fundamental variables present are the angle of the pendulum and the rate of change of angle (i.e. rotational velocity of the pendulum). Moreover, we note that the angle is naturally restricted to be in the domain $[0, 2\pi)$ while the velocity is restricted to be in the range $[-8, 8]$ by most authors describing the problem.

Furthermore, valid actions are members of the real numbers and reflect the torque generated by the the motor at the rotation point of the pendulum. Again, a restriction is put in place by authors formalising the problem restricting the torque to the range $[-2, 2]$.

With now formalise these descriptions to define both the state space the action space for the inverted pendulum control problem.

Definition 5.8 (State space for inverted pendulum)**.** For the inverted pendulum problem, the state space is the set $S \subseteq \mathbb{R}^2$ defined as

$$S = \{(\theta, \dot{\theta}) \mid \theta \in [0, 2\pi], \dot{\theta} \in [-8, 8]\}$$

Definition 5.9 (Action space for inverted pendulum)**.** For the inverted pendulum problem, the action space is the set $A \subseteq \mathbb{R}$ defined as

$$A = [-2, 2]$$

With these foundational definitions out of the way, we move on and define the more interesting agent and environment networks. We start with agent networks as we have already discussed the neural network underpinning the agent network in the previous section.

We see that the neural network controlling the pendulum takes the cosine and sine of the angle rather than the raw angle. This is because it has been experimentally shown that, for various reasons unimportant to this project, training networks to solve the problem is much easier when this is the case. The network also takes as input the velocity of the pendulum.

As for the output of the network, the result is a single real number. This value is not clamped within the network itself so our output function has to perform this clamping.

We now formalise these principles with the definition of the agent network.

Definition 5.10 (Agent network for inverted pendulum)**.** For the inverted pendulum problem, given the structure of the FFNN N defined in Chapter 3, with the sets S and A from Definitions 5.8 and 5.9 define the following functions

- The function $s : S \to \mathbb{R}^3$ defined as

$$s(\theta, \dot{\theta}) = (\sin\theta, \cos\theta, \dot{\theta})$$

- The function $a : \mathbb{R} \to A$ defined as

$$a(t) = \text{clamp}(t, 2, -2)$$

where the function clamp is defined as

$$\text{clamp}(v, max, min) = \begin{cases} v & \text{if } min \leq v \leq max \\ max & \text{if } v \geq max \\ min & \text{if } min \geq v \end{cases}$$

Then the agent network for the problem is the tuple (N, S, A, s, a).

Now that we have defined agent network, we move on to consider the corresponding environment function. First, we briefly discuss the physics underpinning the environment. This is given by the following equations

$$\dot{\theta}' = \dot{\theta} + \Big(\frac{3g}{2l}\sin\theta + \frac{3t}{ml^2}\Big)dt$$
$$\theta' = \theta + \dot{\theta}'dt$$

where θ' and $\dot{\theta}'$ are the new value of the environment variables and θ and $\dot{\theta}$ are the existing values of the variables. t is the torque provided by the agent controlling the motor. Moreover, the remaining symbols are all constraints with the following meanings

- dt is the time-step taken between evaluations of the environment
- g is the gravitational field strength (≈ 9.8 on Earth)
- l is the length of the pendulum
- m is the mass of the pendulum

These equations map quite straight-forwardly to a definition for the environment function noting that we need to clamp the input.

Definition 5.11 (Environment function for inverted pendulum)**.** For the inverted pendulum problem, we have that the environment function is defined as follows:

$$e((\theta, \dot{\theta}), a) = (\theta + t(\theta, \dot{\theta})dt, t(\theta, \dot{\theta}))$$

where $t(\theta, \dot{\theta}) = \dot{\theta} + \Big(\frac{3g}{2l}\sin\theta + \frac{3t}{ml^2}\Big)dt$.

Finally, we need to consider how the network systems may be encoded as a set of linear constraints. As we discussed earlier in this chapter, this entails showing that the agent network's "glue" functions and the environment function can both be written as a piecewise-linear composition of non-linear functions.

Consider the glue functions s and a of the agent network first. We have that

$$s(\theta, \dot{\theta}) = (\sin\theta, \cos\theta, \dot{\theta})$$

Clearly, the sine and cosine of the angles are non-linear functions. As discussed in the previous chapter, the approach we have adopted in this project is to train FFNNs which can approximate these two functions to a sufficient level of accuracy.

We then invoke Definition 4.2 which shows us that as this function is a linear function of distinct non-linear components (i.e. the sine and cosine), if we train FFNNs for the non-linear functions, we can encode the entire function using a set of linear constraints given by the definition.

Similarly, we see that the environment function also has a non-linear function namely the sine function again. We again invoke the same definition as above to give us the result that this function can also be represented as a set of linear constraints.

With these two results, we can now use Definition 5.7 to derive a set of linear constraints to encode the network system as a whole. This concludes our case study; these sets of linear constraints will prove crucial in the next chapter when we investigate how we can use them to actually perform temporal verification of network systems.

In this chapter, we gave a novel way of combining agent neural networks with the environment that they interact with. We then showed how we could encode a network system using a set of linear constraints. We finished by demonstrating in detail how these definitions may apply to the problem of the inverted pendulum.

6 Verifying network systems with LTL

In the previous chapter, we defined a mechanism by which can consider environment and agent as a single system which we defined as a network system. In this chapter, we move on to talk about how we can perform verification on this new system. However, it is important to see why this new area of verification is necessary considering the work we have already done in Chapter 3.

Reachability specifications over neural networks acting as agents were defined in Chapter 3. While immensely useful for locating bugs in the network, they do not demonstrate the full power of verification. Reachability only allows us to determine the actions of the network "one time-step" in the future; that by defining a reachability specification and verifying it, we can only ensure that the network does what we are expecting in that limited scenario. It does not tell us much about the correctness of the evolution of the system over time.

In general, we would want to considReachability specifications over neural networks acting as agents were defined in Chapter 3. While immensely useful for locating bugs in the network, they do not demonstrate the full power of verification. Reachability only allows us to determine the actions of the network "one time-step" in the future; that by defining a reachability specification and verifying it, we can only ensure that the network does what we are expecting in that limited scenario. It does not tell us much about the correctness of the evolution of the system over time. er the actions generated by this agent neural network in the context of the states which result from them rather than reasoning about the actions directly. That is, we wish to include a *temporal* element to our analysis by considering how our network behaves over a period of time.

This idea is well known in the field of systems verification as we discussed in Chapter 2 and the language LTL described there is ideally suited to describing the type specifications we would wish to verify. Therefore, instead of inventing our own modelling language from scratch, we try to reuse as much of the terminology and syntax from existing research as possible.

We will start by defining an interpretation of a Kripke frame for network systems. We

quickly discuss explicit LTL verification and its shortcomings. We then define the concept of linear approximations and show how we can generate these approximations for state sets generated by neural networks. We also give an intuition of how we can generate more complex constraints as well.

We present an algorithm which can be used to check LTL specifications on network systems. We conclude by considering a case study as to how we can apply the techniques we describe to the pendulum balancing control problem.

As with much of the previous chapters, our work here is entirely novel to the extent where we are the first to even consider the idea of how temporal verification could be done on neural networks. Thus by developing an entirely new, sound technique which is also reasonably efficient we have advanced the state of art significantly.

One notable point is how our LTL verification algorithm is not just novel for the domain it is applied in but also in the way it operates. In general, LTL verification algorithms work in a manner which "labels" states independently of the formulae they are verifying. That is, the formula being verified does not affect the control flow of how the labelling algorithm functions. On the other hand, with our algorithm, the control flow is intrinsically tied to the exact structure of the formula.

Another point of note is that we believe that the algorithm which we present at the end of the chapter is actually not limited to working on network systems specifically. In fact, we believe that any transition function which could be encoded using a set of linear constraints (and given our work in the last chapter, we believe that this is possible for an arbitrary function), this algorithm can be used to check LTL formulae. As this is not directly relevant to this project we will not discuss this further but we believe this makes the algorithm much more significant.

6.1 Explicit LTL verification

As we outlined in the introduction, LTL is based on Kripke frames and models. By defining an interpretation of Kripke frame on network systems, we obtain the semantic interpretation of LTL for free on network systems.

Definition 6.1 (Kripke frames on network system)**.** Let $NS = (G, E, t)$ be a network system. Let further S be the set of possible states of the environment in NS.

Then we define the Kripke frame of NS as $F_{NS} = (W, R)$ where $W = S$ is the set of

worlds and $R = \{(w, t(w)) \mid w \in W\}$ is the relation between worlds.

In other words, the set of worlds of the Kripke frame defined by a network system is precisely the set of states on which the agent and environment networks are defined one. Moreover, the relation between the worlds is simply defined by the transition function defined by the combination of the agent and environment networks.

This elegant formulation is quite natural and leads to a simple definition of Kripke frames. Moreover, we have clearly have that R is a serial relation. This is because t is a total function so for each $w \in W = S$, we have that $(w, t(w)) \in W$ which is precisely what is required for a serial relation.

By virtue of R being serial, we have that a Kripke model $M = (F, \pi)$ for any $\pi : P \to 2^W$, is a valid LTL model as well. This means that we can define specifications and interpret them on the derived models.

We start our work by presenting an explicit algorithm to perform LTL verification on network systems. This gives us a baseline from which to measure future work as well as providing the reader with a basic understanding of how LTL verification would work in the simplest case.

Algorithm 6.2 gives the full explicit algorithm relying on Algorithm 6.1 to perform the verification for each state.

One should be able to see straight away how this solution is not at all scalable. Verifying the formula on individual points is clearly not a technique which could work with anything more than a handful of points. The main use for this algorithm is to serve as a way to easily understanding the principles behind LTL verification rather than as a practical algorithm. Thus, we do not discuss explicit verification any further.

6.2 Linear approximations of state sets

As we have seen in the previous section, explicitly verifying LTL formulae is very inefficient and does not take into account the fact that the set of starting states for verification are usually concentrated around a certain point.

The main issue with the algorithms above are that starting states are treated independently of each other rather than being considered as one big group. In this section, we seek to introduce some definitions and notation which will allow us to handle the starting sets (and their respective successors in time) as groups of states making verification

Algorithm 6.1 Explicitly verifies the LTL formula f on the Kripke Model M_{NS} for network system NS given the starting state s

```
1: procedure VERIFY_LTL(M_NS, s, f)
2:     π = Value_Fn(M_NS)
3:     t = Transition_Fn(M_NS)
4:     switch f do
5:         case p
6:             return s ∈ π(p)
7:         case ¬f'
8:             return ¬ Verify_LTL(M_NS, s, f')
9:         case a ∨ b
10:            return Verify_LTL(M_NS, s, a) ∨ Verify_LTL(M_NS, s, b)
11:        case Xf'
12:            return Verify_LTL(M_NS, t(s), f')
13:        case aUb
14:            i = 0
15:            s' = s
16:            seen = {}
17:            while ¬ Verify_LTL(M_NS, s', a) do
18:                seen = seen ∪ {s'}
19:                s' = t(s')
20:                i += 1
21:                if s' ∈ seen then
22:                    return false
23:                end if
24:            end while
25:            s' = s
26:            for i = 0 to i do
27:                if ¬ Verify_LTL(M_NS, s', a) then
28:                    return false
29:                end if
30:                s' = t(s')
31:            end for
32: end procedure
```

Algorithm 6.2 Explicitly verifies the LTL formula f on the Kripke Model M_{NS} for network system NS given set of starting states S

```
1: procedure VERIFY_LTL_SET(M_NS, S, f)
2:     for s ∈ S do
3:         if ¬ Verify_LTL(M_NS, s, f) then
4:             return false
5:         end if
6:     end for
7:     return true
8: end procedure
```

more efficient.

As we have until now worked with linear constraints and continue to seek problems which can be solved with an LP solver, we will define an approximation for the set of starting states using linear constraints.

Before proceeding further, it is instructive to consider why these approximated sets are needed. While we have discussed why it is more efficient to consider sets of states rather than individual states, one may also wonder why we do not use the sets directly themselves. This is best explained through the use of an example.

Suppose our state set is $S = \mathbb{R}$ and our action set is also $A = \mathbb{R}$. Suppose that we have a network system NS with transition function $t : S \times A \to S$. We have already shown in the previous chapter that we can represent the network system as a set of linear constraints; this set of constraints captures the semantics of the transition function of NS.

Let C_{NS} be this set of linear constraints with c_{in} be the LP variable representing the current state of the system and c_{out} be the LP variable representing the new state of the system.

To begin, our system will be in some set of starting states: take this to be the set $[-1, 1]$ for example. To represent this within the set of constraints, we could add the constraints $-1 \leq c_i n$ and $c_i n \leq 1$ to C_{NS}. It may not be completely obvious, but now C_{NS} implicitly represents a set; we constrain the variable c_{out} to only take values which can be reached from the network system given the constraints of the starting states we defined above.

Now suppose that we want to take another step forward using the transition function.

We again add a copy of the constraints in the original set C_{NS} with fresh LP variables c'_{in} and c'_{out}. Moreover, we also need to add the constraint $c'_{in} = c_{out}$; that is that the output of the first step is the new input of second step.

More generally, if we want to take n steps forward from the initial state, we would need to add n copies of the constraint set and connect up the output LP variable of each set to the input of each other.initial

However, we note now that this is not a solution which scales. The size of C_{NS} is dependent on several factors with the main ones being the size of the agent neural network and the number of neural networks used in the encoding of the environment function. As we noted in Chapter 3, the number of constraints has a significant effect on the speed of solving linear programs. Thus repeatedly adding large sets of constraints to LPs would cause any verification technique based on this idea to be very slow.

To solve this issue, in this chapter, we consider the idea of approximating these sets. Instead of representing defining the state set exactly using the constraint set of the network system, we can instead generate a small number of linear constraints which approximate this constraint set well.

Then, when considering a state set n steps forward from the initial state set, we can instead take one step forward and at each step approximate the resulting state set. We then only need to add these small number of linear constraints when taking the next step. We then repeat, each time when taking a step, constraining the input using the approximated state set, taking the output and approximating the result.

To define the process of approximation, we will assume from now on that the set of states S is a real vector. That is we assume that $S = W \subseteq \mathbb{R}^s$ for some $s \in \mathbb{R}$. This is very similar to the assumption we made in the previous chapter to allow for us to define sets of linear constraints on agent networks and environment functions.

> **Definition 6.2** (Linear approximation of a set)**.** Let $S \subseteq \mathbb{R}^s$ be an set. A linear approximation of S is any set S such that $S \subseteq A$ and S is a linearly definable set with constraints A_c.

We see that this definition corresponds with the idea of "over-approximation" from static program verification. We see that states which are not in our initial set of states may be included but we also see that certainly every state in our starting set is also included. This will allow us to define a sound but (usually) incomplete algorithm in the next chapter.

We now present an algorithm utilising solutions of linear programs which allows us to

compute the most basic linear approximation of sets of real vectors. We then enhance this algorithm by making the approximation more accurate by introducing a novel way to add additional constraints.

Algorithm 6.3 Algorithm for generating a coarse linear approximation to a state set

```
1: procedure SIMPLE_APPROXIMATE_STATES(S_constraints)
2:     constraints = {}
3:     for dimension = 1 to s do
4:         max[dimension] = Boundary_Point(S_constraints, MAXIMIZE)
5:         min[dimension] = Boundary_Point(S_constraints, MINIMIZE)
6:         constraints = constraints ∪ {s*_dimension ≥ min[dimension]}
7:         constraints = constraints ∪ {s*_dimension ≤ max[dimension]}
8:     end for
9:     return constraints
```

Algorithm 6.3 gives the method to compute this set. Instead of computing the actual set itself which would be essentially impossible to enumerate, it computes the constraints which define the boundary of the set.

As we can see, the algorithm defines an extremely coarse boundary of the set. It simply computes the maximum and minimum points in each direction and then defines a constraint for each of these computations. In essence, the algorithm defines an s-dimensional hyper-cuboid which encloses all the points in the state set.

While this approximation could be used directly in the verification algorithm we present, in practice the hyper-cuboid includes a lot of "spurious states". This is because the set being approximated is usually a complex (sometimes convex) set with many sides and faces.

Thus while the approximation captures the set, it also includes the states which are outside the set. If too many of these are included, while our verification remains sound, it would be unable to verify positively in most cases instead indicating that we cannot perform verification in either direction.

For this reason, we want as tight an approximation of the true set as possible. We wish to stick with linear constraints for reasons we have already outlined several to this point. Moreover, in general, linear constraints would allow for an arbitrarily close approximation to a convex set so are more than sufficient for our purposes.

However, while we want the approximation to be good, we also need to be wary of the fact that finding these boundary constraints also would get increasingly difficult as we

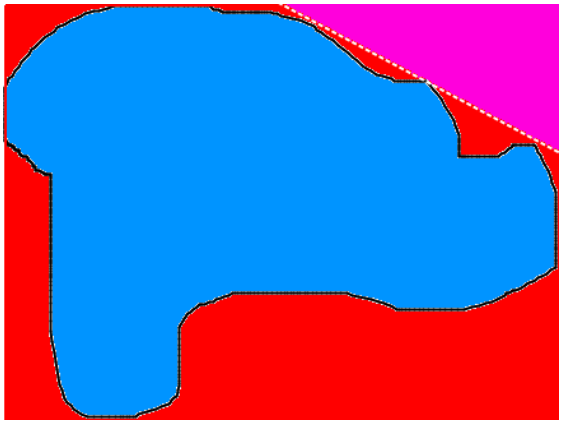


Figure 6.1: Representation of a state set in 2-D.

try to make it more accurate. Moreover, adding these constraints to the final verification problem also increases the time here.

For this reason, we present an algorithm which only adds one extra constraint for each corner of the hyper-cuboid. The allows the algorithm to balance these two concerns while also being easy to understand. We believe there is a lot of potential for additional investigation here as we have only scratched the surface of what is possible.

The first action performed by the algorithm is to find the set of points which define the boundary of set of states. This simply utilises the algorithm we presented earlier in this section; finding the hyper-cuboid enclosing the set implicitly requires finding the boundary points.

Once we have these boundary points, we need to find additional hyper-planes which remove the spurious states which we discussed earlier. To do this, we consider all possible pairs of boundary points in different dimensions. We then try to find a hyper-plane between these two dimensions which would essentially remove as many spurious states as possible while retaining all valid states.

Describing this process in n-dimensions is very tricky so we consider instead two dimensions which makes this process much easier to understand. Suppose we have a set represented in red by Figure 6.1. We see that the parts in blue are members of the true state set which are approximating. The sides of the rectangle give the boundaries of the approximated set. As we can see, there are a lot of spurious states present in this approximation given by the portions marked in red.

To remove some of these red states, we see that we could add a constraint represented

by the dashed line in the diagram. By adding this constraint, we do not remove any valid states but remove a large number of spurious states represented in pink. The key question remains: how do we find this line?

Upon inspection, we see that there are two important properties that any sound, optimal (i.e. excluding the greatest number of spurious states) line which can act as a constraint must have

1. For soundness: the line must have all the points in the set on the same side. That is that valid states must lie entirely to the left or right of the line.

2. For optimality: the line must pass through at least one point in the set. If this was not the case, then we could draw a line which excluded more spurious states while still being correct.

These two properties are the ones we used when building the algorithm to find the new constraint line. As we endeavour to keep working with linear optimization, we want to turn the problem of finding the line into a linear maximisation or minimisation.

To do this, we note that finding the optimal line can be represented as maximising the distance between the points at which the line touches the sides of the rectangle and the boundary points; these distances are represented by the two double headed arrows on the diagram. We also have the constraint that there exists a point in the set such that the line passes through it.

However, if we were to simultaneously maximise the distance in both dimensions, the problem would turn into a quadratic one. As we wish to keep solving linear problems, we need to instead fix a gradient for the line and maximise in only one dimension. The natural gradient to choose here is the one between the two boundary points.

All of this gives us the following definition for the LP to solve to generate the equation of the line.

Definition 6.3 (LP to find linear approximation constraints)**.** Let $S \subseteq \mathbb{R}^2$ be a set of states we wish to approximate defined by the set of MILP constraints S_c defined on variables s_1^* and s_2^*.

Suppose $b_1 = (x_1, y_1) \in \mathbb{R}^2$ is a boundary point for the first dimension and $b_2 = (x_2, y_2) \in \mathbb{R}^2$ is a boundary point for the second dimension. Then, take $m = \frac{y_2 - y_1}{x_2 - x_1}$ as the gradient of the line joining the two boundary points.

Then, solving the following MILP can be used to find a constraint to approximate S:

$$\text{maximize } d^* \text{ where}$$
$$S_c \text{ holds}$$
$$s_2^* - (y_1 + sign * d^*) = m(s_1^* - x_1)$$

where *sign* is a constant with value 1 if the line passes right/above the boundary point (x_1, x_2) and -1 if the line passes left/below the boundary point.

Fundamentally, this LP is attempting to figure out one thing: the most extreme point in the set which the line with gradient m passes through. This is represented by the value of the variables (s_1^*, s_2^*) in the LP. With this point, we can define the actual constraints which can be added to the improve the approximation of the set.

Proposition 6.1 (Linear approximation constraints)**.** *Let $S \subseteq \mathbb{R}^2$ be a set of states we wish to approximate.*

Suppose $b_1 = (x_1, y_1) \in \mathbb{R}^2$ is a boundary point for the first dimension and $b_2 = (x_2, y_2) \in \mathbb{R}^2$ is a boundary point for the second dimension. Then, take $m = \frac{y_2 - y_1}{x_2 - x_1}$ as the gradient of the line joining the two boundary points.

Let L be the LP from Definition 6.3 defined using the above boundary points. Let d be the value of the objective given by the optimal solution for L. Further let (s_1, s_2) be the values of the variables s_1^* and s_2^* in the solution.

Define the line L as $y - s_2 = m(x - s_1)$. Then, a constraint which can be used to approximate S using the variables x and y depending on the location of the boundary points with respect to the line is as follows

1. *If b_1 and b_2 are above L, then the constraint is $y - s_2 - m(x - s_1) \geq 0$.*
2. *If b_1 and b_2 are below L, then the constraint is $y - s_2 - m(x - s_1) \leq 0$.*

We first present a sub-algorithm which composes the work we have done to find the boundary points and the additional approximation constraints.

Algorithm 6.4 gives the routine to generate the set of constraints which approximate a set. It works by first finding the boundary points for each dimension, adding constraints representing the bounding hyper-planes for each of these boundary points.

Then, for each pair of distinct dimensions, it adds four constraints representing the extra hyper-planes which we generate using the solution to the LP which we discussed in the

Algorithm 6.4 Generates a set of linear constraints which approximate the set $S \subseteq \mathbb{R}^s$ defined by constraints *S_constraints*

```
1: procedure APPROXIMATE_STATES(S_constraints)
2:     constraints = {}
3:     for dimension = 1 to s do
4:         max[dimension] = Boundary_Point(S_constraints, MAXIMIZE)
5:         min[dimension] = Boundary_Point(S_constraints, MINIMIZE)
6:         constraints = constraints ∪ {s*_dimension ≥ min[dimension]}
7:         constraints = constraints ∪ {s*_dimension ≤ max[dimension]}
8:     end for
9:     for i = 1 to s do
10:        for j = i + 1 to s do
11:            c_max_max = Linear constraint (for 2-D from Proposition 6.1
   using s*_i and s*_j as variables) with boundary points max[i] and max[j]
12:            c_max_min = Linear constraint (for 2-D from Proposition 6.1
   using s*_i and s*_j as variables) with boundary points max[i] and min[i]
13:            c_min_max = Linear constraint (for 2-D from Proposition 6.1
   using s*_i and s*_j as variables) with boundary points min[i] and max[j]
14:            c_min_min = Linear constraint (for 2-D from Proposition 6.1
   using s*_i and s*_j as variables) with boundary points min[i] and min[i]
15:            constraints = constraints ∪ {c_max_max, c_max_min, c_min_max,
   c_min_min}
16:        end for
17:    end for
18:    return constraints
19: end procedure=0
```

previous section of this chapter. At the end of the routine, it finally returns the set of constraints.

6.3 Checking state subsets

One last piece of the puzzle we need to present before giving our final verification algorithm is how to check if one linearly definable set is a subset of another. This check is a vital part of our final algorithm and it is important that it is efficient as it can be used many times in the verification of one formula.

More formally, what we are trying to check is if two sets X and Y are both linearly definable, then is it the case that $X \subseteq Y$. This is equivalent to the statement for all $x \in X$ then $x \in Y$. As linear programs work best with existential statements rather than for all statements, we need to turn this statement into a counter-example guided one.

That is, we seek to find an $x \in X$ but $x \notin Y$. This is the same as asking if there exists a $x \in X$ and $x \in Y^c$ where Y^c is the complement of Y.

To linear program which is able to determine this, we first need to define a set of linear constraints which encode Y^c. We do this using the following definition.

Definition 6.4 (Linear constraints encoding complement)**.** Let $X \subseteq \mathbb{R}^m$ be a linearly definable set and let C_x be a set of linear constraints encoding X.

For each constraint c in C_x, we define the inverse constraint(s) as

- If $c \equiv f(x) \leq c$ (i.e. a less than constraint) then $f(x) \geq c - Mz$
- If $c \equiv f(x) \geq c$ (i.e. a greater than constraint) then $f(x) \leq c + Mz$
- If $c \equiv f(x) = c$ (i.e. an equality constraint) then $f(x) \geq c - Mz$ and $f(x) \leq c + Mz'$

where z and z' are a fresh LP binary variable for each constraint and M is a "sufficiently large" constant. Let $Z = \sum$(all fresh binary variables z) for all inverse constraints of those in C_x.

We define the set of *linear constraints encoding the complement* of X (i.e. X^c) as

$$C_{x^c} = \{\text{complement of } c \mid c \in C_x\} \cup \{Z \geq 1\}$$

The intuitive idea of how this definition works is a lot simpler than the actual details. The key to understanding it relies on the fact that if some x is in the set, then it satisfies every constraint in the set. So for any x not in the set, it must not satisfy at least one constraint. That is, it satisfies at least one negation of each constraint.

Therefore, to encode the complement, we simply need to check if any of the negation of the constraints are satisfied. Doing this is a bit tricky in a linear program as by definition every constraint must be satisfied in such a program. However, we can use similar tricks as in Chapter 3 to encode either-or constraints.

By introducing the fresh z variables, we essentially allow that constraint to not be satisfied. We however, require at least one of these constraints to be satisfied by the constraint $Z \geq 1$. Thus, we efficiently encode the complement using only a small number of extra binary variables; we expect the number of linear constraints encoding a set to be quite small in most cases.

We formalise this as a linear program using the following definition.

Definition 6.5 (Linear program to check subsets)**.** Let $X \subseteq \mathbb{R}^m$ and $Y \subseteq \mathbb{R}^m$ be linearly definable sets. Let C_x be a set of linear constraints defining X. Let further C_{y^c} be the set of constraints encoding the complement of Y **defined using the same variables as those used in** C_x.

Then we define the following *LP to check if X is a subset of Y*:

$$\begin{aligned} &\text{minimize } z = 0 \text{ such that} \\ &C_x \text{ holds} \\ &C_{y^c} \text{ holds} \end{aligned}$$

We conclude with the theorem which formally ties this LP with the semantics that it defines.

Proposition 6.2. *Let $X \subseteq \mathbb{R}^m$ and $Y \subseteq \mathbb{R}^m$ be linearly definable sets. Let L be the LP defined to check if X is a subset of Y.*

Then we have that the following statement holds: L has a feasible solution x if and only if $X \not\subseteq Y$.

6.4 LTL Verification using approximations

We now present the complete algorithm which allows us to perform LTL verification using the set approximation techniques.

We do, however, need one more restriction for this algorithm to work as intended; for any Kripke Model M, we will assume that any set in the image of π is linearly definable. That is, for all $x \in P$, $\pi(x)$ is a linearly definable set. We impose this restriction as the algorithm we present requires that we are able to encode these sets as a set of linear constraints.

Algorithm 6.5 gives the final algorithm for LTL verification using approximations. Fundamentally, most of the algorithm is actually quite simple and can be understood simply by examining it. The complex cases involve the temporal operators where more subtle reasoning is required. We will discuss each of the cases however and give an intuition of each.

We begin with atoms. The idea behind verification of atoms is quite simple: we extract the value function from the model of the network system and then check if the current state set is a subset of the set of states where the atom is true.

We do this by delegating to a function `Is_Subset` with the two sets of constraints. This is valid as we assumed that the set of states where an atom is true is linearly definable. we have not defined `Is_Subset` but the code for which is essentially an algorithmic version of Definition 6.5 and Proposition 6.2.

Next, moving to both negation and logical-or formulae. In both cases, we first recursively verify the sub-formula(e) and check if they return `UNKNOWN`. If this is returned, then it is simply returned out of the function. Otherwise the appropriate logical operation is performed on the results of recursion.

The first interesting case is given by the X operator. Here, we come across our first use of state transitions and approximation techniques. We begin by extracting the transition function constraints from the model and union the constraints on the variables representing the input of the transition function. This gives us a set of constraints which define the set one step in the future.

We proceed to approximate this set using the algorithm we defined earlier in this chapter and then recursively verify the sub-formula. By doing this, we are exactly verifying the sub-formula one step in the future, capturing the semantics of X.

Algorithm 6.5 Check the LTL formula f on the network system Kripke model M_{NS} given the set of starting sets encoded by the constraints C_s

```
1: procedure VERIFY_APPROX_LTL(M_NS, C_s, f)
2:     switch f do
3:         case true
4:             return TRUE
5:         case p
6:             π = Value_Fn(M_NS)
7:             return Is_Subset(C_s, Get_Constraints(π(p)))
8:         case ¬f'
9:             result = Verify_LTL(M_NS, C_s, f')
10:            if result == UNKNOWN then
11:                return UNKNOWN
12:            end if
13:            return !result
14:        case a ∨ b
15:            first = Verify_LTL(M_NS, C_s, a)
16:            second = Verify_LTL(M_NS, C_s, b)
17:            if first == UNKNOWN || second == UNKNOWN then
18:                return UNKNOWN
19:            end if
20:            return first || second
21:        case Xf'
22:            C_next = Network_System_Constraints(M_NS) ∪ C_s
23:            C_next_approx = Approximate_States(C_next)
24:            result = Verify_Approx_LTL(M_NS, C_next_approx, f')
25:            if result == FALSE then
26:                return UNKNOWN
27:            else
28:                return result
29:            end if
30:        case aUb
31:            return Verify_Approx_LTL_Until(M_NS, C_s, a, b)
32: end procedure
```

Algorithm 6.6 Check the LTL formula aUb on the network system Kripke model M_{NS} given the set of starting sets encoded by the constraints C_s

```
1: procedure VERIFY_APPROX_LTL_UNTIL(M_NS, C_s, a, b)
2:     seen_state_sets = []
3:     C_next_approx = C_s
4:     while true do
5:         b_holds = Verify_Approx_LTL(M_NS, C_next_approx, b)
6:         if b_holds == UNKNOWN then
7:             return UNKNOWN
8:         else if b_holds == TRUE then
9:             return TRUE
10:        end if
11:        a_holds = Verify_Approx_LTL(M_NS, C_next_approx, a)
12:        if a_holds == UNKNOWN || a_holds == FALSE then
13:            return UNKNOWN
14:        end if
15:        C_next = Network_System_Constraints(M_NS) ∪ C_next_approx
16:        C_next_approx = Approximate_States(C_next)
17:        for seen in seen_state_sets do
18:            if Is_Subset(C_next_approx, seen) then
19:                return FALSE
20:            end if
21:        end for
22:        seen_state_sets.push(C_next_approx)
23:    end while
24: end procedure
```

Since we are over-approximating the next step state-set, if the formula is verified to be false, we cannot actually predict if this was because the formula was false on a spurious state or truly false. Therefore, we return `UNKNOWN` instead of propagating `FALSE`. In any other case, we return the result on the sub-formula.

Finally, we finish with the U operator. This is by far the most complex of all the operators and has several subtle ideas incorporated as part of the algorithm.

We begin by setting up some variables in preparation for an infinite loop. Then at every iteration, we first check if b holds. If it does or is it is unknown, then as we have already verified that a holds until this point, we return `TRUE` or `UNKNOWN` as appropriate. Next, we check if a holds. Again, due to over-approximation, if we get a negative (or unknown) result, we have to report `UNKNOWN`.

We finish with the most complex part of the algorithm. We begin by computing the approximated successor state set as we did with the X operator. But after we do this, we utilise the idea of subsets and fixed/periodic points. We describe the concepts utilised informally here but more formal definitions are possible.

The idea is that if our new state set is a subset of a state set which verification has already performed over, then we have come across a set which maps to a subset of itself after several iterations. Because all our approximations are over-approximations, it must be the case that the true sets also exhibit this behaviour.

Because of this, performing any further verification is meaningless as we have already verified for a super-set that b does not hold and since the transition function is periodic, we will never find a set where b can be verified to be true. Because of this, we conclude that aUb must be false as b never holds. If we do not find a subset, we continue looping, adding our current set to the list of seen state sets.

Overall, this algorithm is a reasonably efficient one which is able to check LTL formulae on network systems. We note that this algorithm is also a sound one; when it returns true or false, which we formalise in in the theorem below.

Theorem 6.1 (Soundness of LTL verification algorithm)**.** *Let M_{NS} be a Kripke model for a network system. Let f be a LTL formula and C_s be a set of constraints defining a linearly definable state set.*

We have that the following statements holds:

$\forall s \in S, (M_{NS}, s) \vDash f$ *if* `Verify_Approx_LTL`(M_{NS}, C_s, f) *returns* `true`

and

$\forall s \in S, (M_{NS}, s) \not\models f$ *if* `Verify_Approx_LTL`(M_{NS}, C_s, fO) *returns* `false`*.*

6.5 Case study: balancing a pendulum

We finish this chapter by discussing how the verification techniques which we laid out can be used to verify useful properties in real-world control problems for which neural networks are used. To do this, we return to our familiar control problem of balancing a pendulum.

In previous chapters, we have discussed the reachability problems that one may want to verify for this problem. We have also discussed how we may create a network system based on an agent network and the environment function for the problem.

We start by considering properties which may be useful to identify for the problem. The one which stands out by far and the one we will focus on this case study is checking if after a certain point, the pendulum is always close to the vertical.

That is, we want to check if there exists a starting state where the controller is not able to stabilise the system and performs large oscillations around the vertical point. In other words, we say that the system is stable if it is the case that the pendulum is forever within the range $-c$ and c radians of the vertically up position.

This c can be any arbitrary number; we choose $c = \frac{\pi}{2}$. Our goal then is to find the widest range of starting states for which it is the case that a stable configuration is eventually reached and this is sustained forever.

This is clearly a property worth verifying as if we are able to prove that this property held for an wide range of starting states, then we are able to show that the agent neural network is fulfilling its intended purpose of stabilising the pendulum vertically; we believe this is the fundamental goal of the network and thus verifying this property would go a long way towards verifying the correctness of the neural network as a whole.

To make the above more formal, we start by defining a Kripke model for this verification task

> **Definition 6.6** (Kripke model for stability of inverted pendulum)**.** Let NS be the network system for the inverted pendulum problem. Let further F be the Kripke

frame for this network system.

Let $P = \{s\}$ be the set of propositional atoms. Then, let $\pi : P \to 2^W$ be defined as follows

$$\pi(s) = \left\{(\theta, \dot{\theta}) \in W \mid |\theta| \leq \frac{\pi}{2}\right\}$$

We define the Kripke model to verify the stability of the inverted pendulum problem as $M = (F, \pi)$.

This model creates an atom s which is true in precisely the states where we consider the system to be "stable" as defined above. We see that the set of states where s is true is a linearly definable set (using the constraint set $\{\theta \leq \frac{\pi}{2}, \theta \geq -\frac{\pi}{2}\}$). Thus, the algorithm we defined in the previous section can be used to check LTL formulae.

We now need to derive an LTL formula which can perform the stability verification we defined above. With a little thought, we see that a suitable formula is FGs. It states that there exists a point beyond which all states are stable. This captures exactly our informal definition above.

We do however need to translate this into a formula which only contains the U operator as we have not defined how to solve F or G directly in our algorithm. Using temporal identities, we find that

$$\begin{aligned} FGs &\equiv (\text{true})U(Gs) \\ &\equiv (\text{true})U(\neg F \neg s) \\ &\equiv (\text{true})U(\neg((true)U(\neg s)))(:= A) \end{aligned}$$

where we have labelled the final formula A for ease of referencing. We see that our algorithm is now capable of checking this formula.

With this, our discussion of the theoretical aspects of this case study are concluded. We move on to talk about how we actually perform this verification in practice. As with the experiments from Chapter 3, we will be using Python and Gurobi for setting up the experiments.

One rather big caveat we wish to mention is that we have not managed to implement the full LTL verification algorithm specified in the the previous section. In fact, we actually do not implement the LTL algorithm at all. Instead, we have created a custom verification routine specialised towards solving just the verification of the FGs formula which we detailed above.

Our reasons for this are purely due to time constraints; we have not had enough time

to fully implement this algorithm and check its correctness. We do believe that as verification of the stable property for the inverted pendulum provides good insight into how the algorithm may run. This is because it involves both of the most commonly used temporal operators so gives reasonably good coverage of the important parts of the algorithm.

The other caveat involves the algorithm to check the G operator. We note that do not use the exact subset checking code that we gave earlier in this section. This is because, we are able to statically verify whether our sets our subsets by simply directly examining the constraints. We will not go into the details of how this works as this is only possible in 2 dimensions but we have used it here as it is simpler to implement and also cheaper as it does not involve solving an LP.

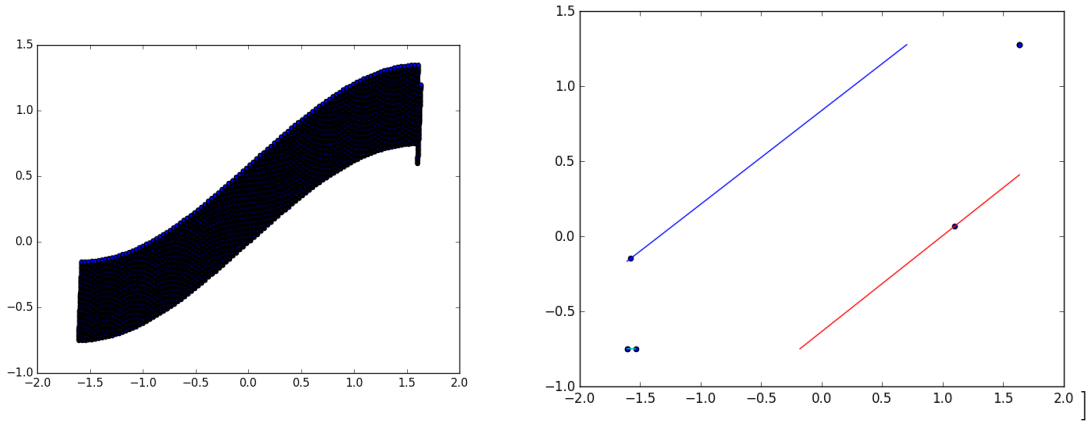
Apart from these two, the remaining parts are implemented in the spirit of the pseudo-code given. This is especially true of the linear set approximation code which has a close to 1-to-1 correspondence with the pseudo-code.

There are two main properties of the algorithm which we wish to evaluate. As well as the obvious analysis of time taken to perform verification, we also want to find a way to check the correctness of the algorithm. A further important point to analyse is whether the additional approximation constraints (i.e. the ones given by Section 4.2) are worth actually including or whether the approximation of the bounding hyper-cuboid (a rectangle in 2-D) is sufficient for our purposes; we consider this as part of both correctness and efficiency.

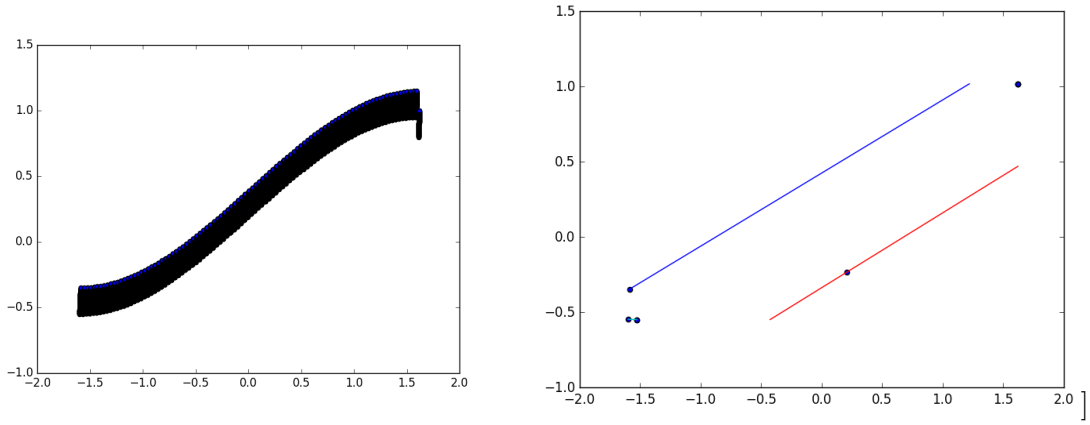
We begin by attempting to show that our implementation is actually a correct one. To do this, we pick a few linearly definable sets of starting sets and use our algorithm to generate an approximation to this set. We compare this approximation to the more traditional approach of selecting a finite set of points from the original set and applying the computed function of the network system to each. By doing this for a large number of points, we can generate a scatter plot.

The aim of this comparison is to check if any points are excluded from the approximation but are present in the plotted set. Unfortunately, we cannot verify correctness better than this because we do not have an alternative verification technique against which we can compare our implementation.

Moreover, we cannot also repeatedly apply the computed function to the points either as the set because too sparse rapidly and gives a very misleading idea of the shape of the set. Because of these factors, we believe this is close to the best we can do to verifying our implementation.



(a) Starting set $\{(\theta, \dot{\theta}) \mid |\theta| \leq \frac{\pi}{2}, |\dot{\theta}| \leq 0.3\}$



(b) Starting set $\{(\theta, \dot{\theta}) \mid |\theta| \leq \frac{\pi}{2}, |\dot{\theta}| \leq 0.1\}$

Figure 6.2: Plots of explicit computation of state set and the corresponding approximation after one application to the specified starting state set.

Starting set	A. C.	Time (s)	Indices	Bounds
$\frac{-4\pi}{64} \leq \theta \leq \frac{-3\pi}{64}, \lvert\dot{\theta}\rvert \leq 0.3$	Absent	0.96	(4, 5)	[-0.20, -0.14], [-0.15, 0.16]
$\lvert\theta\rvert \leq \frac{\pi}{64}, \lvert\dot{\theta}\rvert \leq 0.2$	Present	0.48	(3, 4)	[-0.07, 0.02], [-0.26, -0.20]
$\frac{-4\pi}{64} \leq \theta \leq \frac{-3\pi}{64}, \lvert\dot{\theta}\rvert \leq 0.3$	Present	0.63	(1, 4)	[-0.19, -0.15], [-0.13, 0.07]
$\lvert\theta\rvert \leq \frac{\pi}{32}, \lvert\dot{\theta}\rvert \leq 0.3$	Present	1.49	(2, 7)	[-0.12, 0.03], [-0.30, -0.21]
$\lvert\theta\rvert \leq \frac{\pi}{16}, \lvert\dot{\theta}\rvert \leq 0.3$	Present	5.69	(1, 19)	[-0.19, -0.04], [-0.26, 0.12]
$\lvert\theta\rvert \leq \frac{\pi}{32}, \lvert\dot{\theta}\rvert \leq 0.6$	Present	2.98	(6, 10)	[-0.15, 0.00], [-0.28, -0.07]
$\frac{-\pi}{3} \leq \theta \leq \frac{1\pi}{8}, \lvert\dot{\theta}\rvert \leq 0.5$	Present	45.21	(20, 23)	[-1.49, 0.96], [-2.46, 0.71]

Table 6.1: Results of experimentation on verifying stability property for the pendulum problem. All results in the table give results where the property was found to be true. For each starting set of states, we give whether the additional constraints were included, the time taken to perform the full verification, the iteration indices and the final bounds on the variables at the end of the problem. By iteration indices, we mean that for index (i, j), we found summarises set at iteration j to be a subset of the state set at iteration i. By final bounds on the variables, we give the maxima and minima for the angle and the angle derivative at the final iteration of the algorithm.

Figure 6.2 gives these plots for a selection of starting sets. We see that in all three cases, it is indeed the case that the approximation covers the entire set. In each case it is clear that at least some extra states are included as expected. We also note that if we had taken only the bounding rectangle, this would have significantly increased the spurious states in every single case.

Overall, the plots show us that our technique appears to be working as intended. However, this does not by any means rule out the presence of bugs as this is only a very shallow check into the algorithm. We would have attempted to find more metrics to check correctness but given time constraints, this was not possible.

Moving on, we now consider the running times of the algorithms with and without the additional approximation constraints for various starting sets. Table 6.1 summarises the results we obtained for a range of different sets.

We note that it was essentially impossible to get any larger a set to coverage for the case where the additional constraints are not present. This strongly indicates that the approximation without the additional constraints is simply too coarse to be of any use and thus it was a very good idea for us to introduce the theoretical foundation behind these constraints.

Moreover, we note that we are able to actually find rather large sets which are able to be verified to be stable. Especially at the end of the table, we see that sets which span a large range of values in both the angle and the derivative all converge to a fixed point where the whole set is in the stable region. Due to this, we conclude that our technique is quite affective in this regard.

The other point to be made is in the space of efficiency. We see that the size of the starting set has a significant impact on the running time of the algorithm. Specifically, we see that doubling the size of the starting set vastly increases the running time in a non-proportional way. However, overall even for the large sets, the running time is very reasonable for an algorithm which is verifying something as significant as stability.

Moreover, significantly, we see that (although not present in the table) this running time increase actually comes from an increase in the time taken to solve the LPs to find the approximation constraints rather than any other part of the algorithm. Thus, we suggest that this is the area where future effort of efficiency gains should be focused.

In this chapter, we consolidated much of the work we have done throughout the project to produce a novel, sound algorithm to verify LTL formulae on network systems. To do this, we first gave the definition of a Kripke frame followed by giving the explicit verification algorithm and explaining its deficiencies. We then gave a innovative way of approximating state sets output from network systems using linear constraints. We then utilised this at the heart of our LTL verification algorithm. We finished by evaluating a partial implementation of the LTL algorithm on the inverted pendulum problem where we found that the performance of the algorithm was reasonable.

7 ReVerify: a library for performing verification of neural networks

In this final chapter on the work we have done in this project, we briefly present a library that we have developed to consolidate some of the patterns and concepts that we have used in this project. We call this library *ReVerify* coming from the longer ReLU verification.

As with all experiments we have performed, the library is written in Python and it depends on the Gurobi library to handle the linear programming solving. Reflecting the structure of the project itself, the library can be split into two main sub-components

- **Converting an FFNN into a set of constraints**: this component handles converting a feed-forward neural network into a set of linear constraints. It does this by exposing methods which add the variables and then the constraints to encode each layer of FFNNs using a layer-by-layer approach.
- **Verification by checking LTL formulae**: this component contains some helper functions which help with verification of LTL formulae. Specially, the library extracts the reusable code from the experiments from Chapter 6. This is namely the generation of additional constraints for the approximation of state sets and the code which checks whether two sets are subsets of one another.

We will discuss each of these briefly in turn.

7.1 Conversion of FFNNs into sets of constraints

As we described above, this component implements the algorithm we described in Section 3 to convert a FFNN into a set of linear constraints.

The code for this component is structured as a Python class. The constructor for the class takes a Gurobi model which is used later when the methods of the class are called.

The constructor also takes a value for the "big M" constant which we used throughout the constraints. This value should be set to the smallest possible value possible but large enough to allow constraints to be falsely satisfied; this value is determined on a problem-by-problem basis.

There are two functions which are intended for public use in the class.

1. **add_vars(layers)** this function takes a list of layers of an FFNN and proceeds to add the all the variables appropriate on the type of layer. The layers expected by the library are data structures defined by the Keras neural network library.

   An important implementation detail to note about Keras layers is that the activation function is actually decoupled from the neural network layer. That is, Keras generates one layer for the linear transformation carried out by each neural network (with an identity activation function) but it further generates another, subsequent "layer" which purely applies the activation function.

   Because of this implementation detail, we distinguish between the two types of "layers" by examining a property which is set by Keras to indicate the activation function.

   The variables we add breaks down depending on the layer as follows

   - **Identity layers**: for layers which perform the linear transformation and have the identity activation function (so called "dense" layers), we add three types of variables to the LP: ϵ, δ and *output* variables. These three types reflect exactly the three symbols used in the theoretical encoding given earlier in the project.

   - **ReLU layers**: for layers which calculate the ReLU of their input (and don't perform any linear operation), we create only a single type $relu$. These are non-negative variables which will represent the output of the ReLU layer.

   As well as adding these variables, we also store them in a list. Since Python is dynamically typed, we can have the represent the identity layers as a tuple of lists of variables and the ReLU layers as a simple list of variables. The function then returns this list to the caller.

2. **add_constraints(layers, input, variables)**: this function takes three parameters with each one representing the following

   - **layers**: this parameter is identical to the list of Keras layers which we de-

scribed for the add_vars(layers) function.

- **input**: this parameter contains an ordered list of the variables used to represent the input to the network as a whole. For example, for the cart-pole problem, this would be the the list $[x, \dot{x}, \theta, \dot{\theta}]$.

- **variables**: this parameter is precisely the output of the add_variables function.

The function then adds the constraints to the LP which correspond to the weights and biases of the layers which were passed in and using the input and network layers given as parameters. The constraints added are precisely those given by our encoding in Chapoter 3.

One may wonder why we split up our function into these two rather than having one big function which carries out everything. This is for the sake of flexibility and because we wish to reflect how Gurobi is used to add variables and constraints.

For flexibility, we see that we can swap or change variables between the variable adding stage and the constraint adding stage. This can include things like modifying the bounds on a variable or changing a variable to a constant for the sake of testing or performance. By splitting the function, we increase complexity slightly but we vastly improve our ability to tweak the LP.

Moreover, this two phase approach also reflects how Gurobi itself operates. Variables need to be added using the "addVariable" function before the "update" function is called and the variable is used in a constraint. By reflecting this, we stick with the existing conventions of Gurobi meaning the user does not need to adapt to the new style if they need to manually add further variables or constraints.

Overall, this class is a relatively small one but extracts all the code common across all the experiments from Chapter 3. We believe it is a useful base on top of which further additions and improvements can be made.

7.2 Verifying network systems by checking LTL formulae

The purpose of this component is to bring together a small set of utility functions which we found to be useful in our experimentation in Chapter 6 and we think may be useful in other problems.

As we discussed in Chapter 6 itself, due to time constraints, much of our work was hard-coded to the exact problem that we were trying to solve. However, there is a small amount of functionality we think is sufficiently general and so worth extracting into this component.

The code for this component also takes the form of a Python class; this class represents a linear approximation of a 2-D state set as we described in Chapter 6.

The constructor for this class takes the four corners of the bounding rectangle of the set and a list of pairs of points. These points are two of the points which lie on the lines delineating the "additional" constraints. Using these parameters, some linear algebra is performed to populate some data structures in the class which will be used by the functions below.

As with the other component, there are two public functions available for use in this class

1. **constraints**: this function takes two parameters; these parameters represent the variables for each dimension of the set which are to be constrained. On these two variables, a list of constraints is generated from the information calculated in the constructor.

   The function returns the list of LP constraints defined on the two variables passed in which restrict the variables to range over the space defined by this set.

2. **is_subset**: this function takes as a parameter another set and computes whether this other set is a subset of this set or not. It does this by considering whether the bounding rectangle of this set contains the bounding rectangle of the other set as well as checking if the constraint points of the other set lie within the correct half-plane defined by the the constraints lines of the current set.

   The function returns true or false depending on whether the passed in set is a subset of the set represented by the object itself.

While there is significant room for increasing the common code in this class and also generalising the code present to n-dimensions once the theory for this is developed, we feel that this class is a solid start towards building a full library to perform checking of LTL formulae on network systems.

In this chapter, we discussed our innovative ReVerify library. We discussed the two

sub-components and detailed the functions in each, how they were used and why they were important.

8 Project Evaluation

In this chapter we evaluate the contributions we have made in this project. Specifically we summarise our theoretical and practical contributions in turn and set out their strengths and weaknesses.

8.1 Theoretical contributions

Each section of the project has brought new theoretical additions but the overarching aim throughout is reflected in the title: verification of ReLU feed-forward neural networks. This began with reachability specifications for these networks followed by considering agent and environment sub-networks forming a system. Finally, we finished with a sound algorithm for verification of LTL on these systems.

In summary, our theoretical contributions are as follows:

1. **Verifying reachability on feed-forward neural networks** We defined the very concept of reachability on neural networks. We encoded ReLU feed-forward neural networks using linear constraints. We proved that our encoding was equivalent to the function computed by the network. We defined a linear program encoding reachability for a ReLU feed-forward network. We proved an equivalence between verifying reachability and solving the linear program. We gave an alternative linear encoding of networks using SOS constraints. We finished with a pseudo-code algorithm which performed reachability verification incoporating changes to control floating point errors.

2. **Network systems** We presented the novel technique of modelling control problems using agent networks and environment functions. We defined how these networks could be composed into a structure called a network system. We discussed how we could turn a network system into a set of linear constraints.

3. **Basic LTL verification on network systems** We presented how Kripke models can be interpreted on network systems. We showed that these models are also valid

LTL models. We presented an algorithm which can be used to perform explicit LTL verification.

4. **Approximation-based LTL verification on network systems** We presented a way to approximate the output of a network system using linear constraints. We presented a linear program which could be solved to generate additional constraints in 2-D. We presented an algorithm which generated the approximation and was complete in 2-D. We finished with an sound algorithm which could perform LTL verification using these approximated sets.

8.1.1 Strengths

We see that our contributions have the following strengths

1. **Soundness and completeness of reachability verification** As we have an equivalence between our linear encoding and the reachability problem the algorithm to verify reachability is both sound and complete. Completeness is an especially valuable property as often verification techniques sacrifice completeness for efficient algorithms. We believe our algorithm is efficient without needing to make this trade-off.

2. **Efficiency of algorithm for LTL verification** When this project was first started, we did not expect to be able to come up with a technique which worked for real world networks. Instead we envisaged that any technique for temporal verification would be restricted to toy networks and much work would be needed to allow this to scale.

   While there are still efficiency concerns especially as the dimensionality of the state space increases, this algorithm is far ahead of our expectations at the start of the project.

3. **Novelty of work** All theory developed was novel. While some early work did overlap with existing ideas and techniques, the exact formulation and all proofs were new. With novelty itself is not something to be intrinsically strived for, we believe we have also made significant progress in solving real world problems and thus the novelty factor is a strength as it indicates the scale of progress.

8.1.2 Weaknesses

We see that our contributions have the following weaknesses

1. **Restriction of portions of approximation algorithm to 2-D** While we firmly believe that our work done in 2 dimensions can indeed be generalised to an arbitrary number of dimensions, we have been unable to show this conclusively due to lack of time to fully formalise this technique.

   We have however, tried to provide the intuition of how such a formalisation may work in our words and this should be possible in future work in this domain. Apart from the small portion on how additional constraints can be generated, the remainder of the work is indeed applicable to state spaces of arbitrary dimensions.

2. **Lack of exploration of properties of network systems** We believe that the formulation of network systems is extremely rich and has much potential for further work. We have only barely scratched the surface of what is possible by providing an algorithm for verification of LTL. We discuss some of these future avenues in the concluding chapter.

3. **Lack of completeness in verification of LTL for network systems** This weakness is a more minor one but the property of completeness would be a desirable one to have although one which we believe is very tricky to obtain. Approximation intrinsically removes any chance of completeness to be possible so another approach would have to be tried.

   Completeness is present in the explicit algorithm: it may be possible to improve this algorithm to increase efficiency while still preserving the property.

8.2 Practical contributions

The sections of the project where we have presented practical applications for our work via pseudo-code algorithms, we have also implemented said algorithms in Python and have performed either one or several case studies to evaluate the practical use of said algorithms. This was the case for both reachability verification and LTL verification on network systems. As well as our evaluation here, we have made more detailed comments throughout the project when discussing these studies.

In summary, our practical contributions are as follows:

1. **Implementation of reachability verification** We implemented the pseudo-code algorithm we presented in Python. We utilised Gurobi to solve the generated LPs. We presented several case studies, both control problems which have been the subject of intense research in recent years but also classification problems where neural networks have traditionally been used. We also gave benchmarks and demonstrated the efficiency of the algorithm.

2. **Proof of concept of verifying LTL formulae using approximations** While not implementing the full LTL algorithm, we did implement significant portions of it including the full approximation code for 2-D. We tested this code by presenting a case study for the inverted pendulum control problem.

   We attempted to demonstrate correctness of our implementation as well as giving a collection of examples where our algorithm was successful in verifying the property of stability. We also demonstrated that the additional approximation constraints are an important addition to have useful verification.

3. **Implemented a library for ReLU FFNN verification named ReVerify** We consolidated some of the useful routines from the code above and produced a library called ReVerify. Specifically, this library contained routines for generating constraints for FFNNs which was used in both reachability verification but also verification of LTL. It also contained an implementation for 2-D state sets used by our LTL work.

8.2.1 Strengths

We see that our contributions have the following strengths

1. **Efficiency of reachability verification** We note that our algorithm for reachability verification was able to solve even rather complex reachability problems on neural networks of a reasonably large size in a very short period of time (i.e. in less than 1 minute).

   While industrial scale networks will be larger and may contain even more complex applications, we think that our algorithm in general would scale well to even these applications given how well it has done with the current examples. This is especially the case as linear programming is inherently very parallelisable so running the LP solver on a cluster of servers rather than a single machine may yield even higher performance benefits.

2. **Strong verification results for LTL verification** We see that the verification results on stability we were able to prove are very strong indeed. We were able to show that, for a large portion of the state space, that the system would always be stable.

   We cannot emphasise how important such a proof is for these sort of applications. As we discussed in the introduction, verification of these networks is a rapidly growing field which will be vital in the future and a technique which is able to prove such strong results will be an important starting point for future research.

3. **Novelty of implementation** Tied closely to the point above on the importance of verification and the novelty of the theory, we again note how all of our implementation was novel.

   In reachability, no implemented technique which converted a FFNN into a linear program to perform reachability verification exists. While techniques which draw inspiration from SAT and happen to use linear programming as part of their technique do exist, we believe our implementation to be competitive with them.

   As for verification using LTL, we are the first who have even considered solving this problem. We have already discussed how our whole theoretical work in the project was novel, but we also stress that we were able to implement an algorithm and demonstrate its usefulness. We believe this significantly advances the state of art in verification of neural networks.

8.2.2 Weaknesses

We see that our contributions have the following weaknesses

1. **Lack of exploration into more complex properties for reachability verification** While we demonstrated the efficiency and applicability of our algorithm through our case studies, we were not able to consider more advanced properties. These include both consideration of metrics such as memory consumption but also investigation into tuning Gurbobi when solving the LPs.

   Gurobi has tens of properties which control how the linear program should be solved; this allows the user to define which technique they consider most optimal so Gurobi can quicker solve the problem. Had we had more time, we would have investigated

Moreover, we also think that a more detailed exploration could be carried out into the use of SOS constraints. As we mentioned earlier, the SOS constraints were generally found to perform worse than big-M constraints but this was not universally the case. We could have investigated this further given more time.

2. **Lack of implementation of full LTL verification algorithm** As we discussed earlier in the project, rather disappointingly, we were unable to implement the full LTL verification algorithm. This was for three main reasons: first our formalisation of the LTL algorithm actually came from our experiments into trying to show stability for inverted pendulum rather than the other way round. Therefore, we actually had the partial implementation before we had the theoretical foundations in place. Due to this, retrospectively modifying the experiments would have been tricky.

   Secondly, we did not have time to actually perform this implementation; the algorithm is surprisingly subtle to implement and would have required a lot of work to make fully correct. Thirdly, we would have needed to find more complex formulae (for example using U) if we wanted to fully test the algorithm which would have taken even longer.

   Due to this, we decided that it was not worth the vast amount of extra time to actual complete the implementation. However, as we said earlier, we believe we have provided a solid foundation on which this algorithm could be developed.

3. **Sparse investigation into LTL verification algorithm** We believe that the portion of the LTL algorithm that we implemented is correct. However, as we mentioned when discussing this, it is very hard to definitively show this. As no techniques exist against which our work can be compared, it is difficult to decide whether our technique is actually working as intended. Given the time, we would have attempted to invent and implement more metrics which would have shown the correctness - perhaps by tracing the paths of boundary points or by comparing to a different abstraction technique.

   Similarly with efficiency, we have tried to provide a range of experiments and given the running time for each. However, it would have been good to carry out a more through investigation to see the scalability of the algorithm to even larger networks and more complex properties.

   Finally, we would have liked to investigate the technique on different types of problems as well but this would have required an extensive amount of work that we did not have time to do.

9 Conclusions and Future Work

Our overarching aim in this project has been to improve the state of the art in the field of verification of neural networks which we believe has been investigated thoroughly enough to date. We believe that we have been largely successful in this endeavour having introduced a wide variety of novel techniques in this domain.

We started by presenting a sound and complete technique to perform reachability verification. We also introduced network systems which allowed for a new method for thinking about how control problems solved using neural networks could be verified.

To allow for network systems to be useful, we provided an interpretation of Kripke semantics over them. Finally, we concluded by introducing methods to approximate output state sets of network systems and presented a sound algorithm which can be used to perform verification of LTL formulae on network systems.

9.1 Summary of work

In this section, we expand on what we give a detailed summary of the entire body of work we have done in this project.

We began by exploring how reachability could be defined on a neural network. We discovered that defining reachability using arbitrary sets made the problem intractable so quickly resolved to working with sets which can be represented using a finite set of constraints.

As we decided early on in the project to work with linear programming rather than more complex constraint solving techniques like SMT, these constraints were all made to be linear. This led us to the formulation of the concept of a linearly definable set which we formally defined. We proceeded by giving a definition of reachability in terms of this concept of linearly definable sets.

We then moved on to show how a neural network could be encoded as a linear program

by building up the definition piece by piece. We started by defining how a single identity or ReLU layer could be encoded by a set of constraints on linear programming variables. We proved that this layer encoding was precisely equal to the function computed by that layer of the network.

We then proceeded to define the encoding of the network as a whole and again prove that this encoding was equal to the function computed by the network as a whole.

We then moved onto the question of reachability itself. We provided a linear program defined on a neural network, an input and output set and proved a theorem which showed that deducing reachability from the input to the output set through the network was equivalent to checking the feasibility of the linear program.

We then gave an alternative encoding using SOS constraints and also solved the practical problem of how to handle floating point approximation errors. We finished by carrying out experiments and benchmarks on a wide variety of real control and synthesised problems and showed the scalability of our technique.

Having tackled reachability, we moved on to consider the problem of temporal verification of networks. To provide a mathematical foundation for this work, we considered how worked on how arbitrary mathematical functions could be turned into a set of linear constraints. We discussed existing techniques and discussed their inadequacies. This motivated our technique of first generating a neural network which represented the function and then using our work on reachability to turn that network into a set of constraints.

We then formalised agent networks and environment functions, somewhat taking inspiration from agent and environment templates in interpreted systems. We proceeded to show how network systems could be turned into a set of linear constraints by utilising our work from previous chapters. We finished with a case study where we demonstrated how we could define a network system for the inverted pendulum problem.

Our next chapter presented how we could verify LTL for network systems. We began by defining how Kripke frames could be produced for an arbitrary network. We then showed how an explicit LTL algorithm could be defined to verify this property. We discussed its inadequacies and motivated the idea of abstracting these state sets using sets of linear constraints. This led us to present a technique which would let us find these sets of constraints and we further refined these sets by using more complex constraints.

We then showed how we could check whether one state set was a subset of another: an important requirement for our final algorithm. We presented a sound algorithm for verifying an LTL formula using the approximation techniques we defined. We finished

by discussing an implementation of this algorithm with the inverted pendulum problem and considered efficiency and correctness of them as well as the impact of the additional constraints.

In our final chapter we discussed the library that we created to aid in verification of ReLU neural networks. Named ReVerify, we discussed how it was broken into two components: one which converted a neural network into a set of linear constraints and the other which generated 2-D approximation sets for state sets and allowed subset checks. We presented in detail the function definitions and the purpose of each of the functions.

9.2 Future extensions

While we have significantly advanced the state of the art of verifying ReLU feed-forward neural networks, this area of research is so rich for exploration that we feel that it will take years of research to fully explore the potential ideas. Throughout the project, we have signposted areas where we think that future research is important. Here, we give a summary of, what we believe are, the most important points which need further research in the two major areas our project touches.

9.2.1 Verifying reachability on FFNNs

- **SOS constraints** While we presented an encoding of FFNNs using SOS constraints and briefly investigated their efficiency, we did not provide an extensive study into their performance characteristics compared to big-M constraints. We think it is important to study this as for certain classes of problems, one encoding or the other may be more efficient and it is important to analyse this split to be as performant as possible when verifying reachability for large networks.

- **Scaling through concurrency and tuning** We showed that our algorithm for verifying reachability was efficient over a large number of networks. However, industrial networks are likely to be larger and require verification of more complex properties so we will need to improve our algorithm to remain efficient given this increased demands.

  One rather elegant way to do this is to simply run the verification over a larger number of machines. Solving integer linear programs is already very parallelisable with Gurobi making use of all the cores of a single machine. By replicating this idea over a cluster of machines, an even faster algorithm may be possible.

Moreover, Gurobi is a very advanced LP solver and contains a large number of parameters and constants which can be tweaked to quicker solve LPs. While we have simply taken the default ones in most cases, tuning these parameters for the reachability problem could yield significant performance benefits.

- **Comparison to other reachability techniques** In the introduction, we discussed two other papers which present techniques which either verify reachability or could be easily modified to do so. One of these papers only presented their results without releasing their code or networks while the other release both of these. The issue was, these papers (especially the second) appeared quite late in the project timeline so we were already investigating LTL verification by this point.

  We believe that it would be worthwhile investigating and comparing our technique against those of these papers,especially the second paper where the networks are available to analyse and utilise our technique. Moreover, ideas from their techniques could also be incorporated into our technique to make our technique more efficient if it is found that theirs are superior.

9.2.2 Verifying network systems using LTL

- **Generalising set approximations** Currently, the algorithm for set approximations are limited to 2-D. We have pointed out how this generalisation can be made by informally describing this general algorithm. However, actually mathematically defining these approximation is crucial to allow the later LTL verification work to be usable for problems where there are more than two state variables.

- **Implementing full LTL verification algorithm** As we discussed when presenting the case study, we tailored our algorithm to verifying just the stability problem for the inverted pendulum problem. We did not have time to implement the full LTL verification algorithm. However, the power of temporal verification using LTL relies on the full power of all the LTL operators. For this reason, we think it is crucial that the full LTL algorithm is implemented and analysed.

- **Investigating correctness and efficiency of LTL verification** While we presented some evidence of the correctness and relative efficiency of our verification techniques for the inverted pendulum problem, we also gave caveats as to why this was insufficient to draw any strong conclusions about the algorithm. It is important that this issue be resolved so that the technique can be accepted and used in a wider range of techniques.

One would need a novel way of checking the correctness of the algorithm; as we have suggested there are currently no techniques against which we can compare our algorithm which makes it difficult to check correctness. For efficiency, more complex properties should be represented with LTL and these should be checked on larger and higher dimensional network systems. By doing this, confidence in the algorithm will be improved increasing its popularity and reach.

Bibliography

[1] A. Barto, R. Sutton, and C. Anderson. "Neuronlike adaptive elements that can solve difficult learning control problems". In: *IEEE transactions on systems, man, and cybernetics* 5 (1983), pp. 834–846.

[2] A. Guez and J. Selinsky. "A trainable neuromorphic controller". In: *Journal of robotic systems* 5.4 (1988), pp. 363–388.

[3] A. Moore. *Efficient memory-based learning for robot control*. Tech. rep. UCAM-CL-TR-209. University of Cambridge, Computer Laboratory, Nov. 1990. URL: http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-209.pdf.

[4] Martin Anthony. *Discrete Mathematics of Neural Networks*. Society for Industrial and Applied Mathematics, Jan. 2001. ISBN: 978-0-89871-480-7. DOI: 10.1137/1.9780898718539.

[5] B. Widrow and F. Smith. "Pattern-recognizing control systems". In: *Computer and Information Sciences (COINS) Proceedings* (1964).

[6] Mike Barnett et al. "Weakest-precondition of unstructured programs". In: *The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '05*. Vol. 31. 1. New York, New York, USA: ACM Press, 2005, p. 82. ISBN: 1595932399. DOI: 10.1145/1108792.1108813.

[7] CW Barrett, R Sebastiani, and SA Seshia. "Satisfiability Modulo Theories." In: *Handbook of* (2009).

[8] Osbert Bastani et al. "Measuring Neural Net Robustness with Constraints". In: (May 2016). arXiv: 1605.07262.

[9] Christopher M. Bishop. *Neural networks for pattern recognition*. Clarendon Press, 1995, p. 482. ISBN: 9780198538646.

[10] Per Bjesse and Per. "What is formal verification?" In: *ACM SIGDA Newsletter* 35.24 (Dec. 2005), p. 1. ISSN: 01635743. DOI: 10.1145/1113792.1113794.

[11] C. Anderson. "Learning to control an inverted pendulum using neural networks". In: *IEEE Control Systems Magazine* 9.3 (1989), pp. 31–37.

[12] C. François. *Keras*. https://github.com/fchollet/keras. 2015. URL: %7Bhttps://github.com/fchollet/keras%7D.

[13] Alastair Donaldson and Christian Cadar. *Software Reliability*. 2016.

[14] PEK Donaldson. "Error decorrelation: a technique for matching a class of functions". In: *Proceedings of the Third International Conference on Medical Electronics*. 1960, pp. 173–178.

[15] E. Beale and J. Tomlin. "Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables". In: *OR* 69.447-454 (1970), p. 99.

[16] E. Beale and J. Tomlin. "Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables". In: *OR* 69.447-454 (1970), p. 99.

[17] Ruediger Ehlers. "Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks". In: *CoRR* abs/1705.01320 (2017). URL: http://arxiv.org/abs/1705.01320.

[18] Jean-Christophe Filliatre. "Deductive software verification". In: *International Journal on Software Tools for Technology Transfer* 13.5 (Oct. 2011), pp. 397–403. ISSN: 1433-2779. DOI: 10.1007/s10009-011-0211-0.

[19] G. Katz et al. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks". In: *ArXiv e-prints* (Feb. 2017). arXiv: 1702.01135.

[20] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. "Learning to Forget: Continual Prediction with LSTM". In: *Neural Computation* 12.10 (Oct. 2000), pp. 2451–2471. ISSN: 0899-7667. DOI: 10.1162/089976600300015015.

[21] A Graves. "Supervised sequence labelling". In: *Supervised Sequence Labelling with Recurrent Neural* (2012).

[22] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. "Speech Recognition with Deep Recurrent Neural Networks". In: (Mar. 2013). arXiv: 1303.5778.

[23] Inc. Gurobi Optimization. *Gurobi Optimizer Reference Manual*. http://www.gurobi.com. 2016.

[24] H. Mittelmann. *Benchmarks for Optimization Software*. http://plato.asu.edu/bench.html. 2016. URL: %7Bhttp://plato.asu.edu/bench.html%7D.

[25] A Hauge and A Tonnesen. "Use of Artificial Neural Networks in Safety Critical Systems". In: *Faculty of Computer Sciences* (2004).

[26] Simon S. Haykin. *Neural networks : a comprehensive foundation*. Prentice Hall, 1998. ISBN: 0780334949.

[27] S.S. Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999. ISBN: 9780139083853.

[28] Sepp Hochreiter and Jurgen Jürgen Schmidhuber. “Long short-term memory”. In: *Neural Computation* 9.8 (1997), pp. 1–32. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. arXiv: 1206.2944.

[29] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. “Multilayer feedforward networks are universal approximators”. In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 08936080. DOI: 10.1016/0893-6080(89)90020-8.

[30] Xiaowei Huang et al. “Safety Verification of Deep Neural Networks”. In: (Oct. 2016). arXiv: 1610.06940.

[31] M Huth and M Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. 2004.

[32] K. Furuta, M. Yamakita, and S. Kobayashi. “Swing up control of inverted pendulum”. In: *Industrial Electronics, Control and Instrumentation, 1991. Proceedings. IECON'91., 1991 International Conference on*. IEEE, pp. 2193–2198.

[33] Andrej Karpathy. *Convolutional Neural Networks for Visual Recognition*. 2016. URL: http://cs231n.github.io/neural-networks-1/%7B%5C#%7Dactfun (visited on 01/26/2017).

[34] Robert A. Kosiński and Cezary Kozłowski. “Artificial Neural Networks–Modern Systems for Safety Control”. In: *International Journal of Occupational Safety and Ergonomics* 4.3 (Jan. 1998), pp. 317–332. ISSN: 1080-3548. DOI: 10.1080/10803548.1998.11076397.

[35] Z Kurd and T Kelly. “Establishing safety criteria for artificial neural networks”. In: *International Conference on Knowledge-Based and* (2003).

[36] Zeshan Kurd. *Artificial Neural Networks in Safety-critical Applications*. Tech. rep. 2002.

[37] Zeshan Kurd, Tim Kelly, and Jim Austin. “Developing artificial neural networks for safety critical systems”. In: *Neural Computing and Applications* 16.1 (Oct. 2006), pp. 11–19. ISSN: 0941-0643. DOI: 10.1007/s00521-006-0039-9.

[38] LISA lab. *LSTM Networks for Sentiment Analysis*. 2016. URL: http://deeplearning.net/tutorial/lstm.html.

[39] Alessio Lomuscio. *Systems Verification*. London, 2016.

[40] M. Plappert. *keras-rl*. https://github.com/matthiasplappert/keras-rl. 2016.

[41] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. Worldwide series in computer science. Wiley, 2006. ISBN: 9780470093559. URL: https://books.google.co.uk/books?id=CpJQAAAAMAAJ.

[42] N. J. Medrano-Marqués and B. Martín-del-Brío. "Sensor linearization with neural networks". In: *IEEE Transactions on Industrial Electronics* 48.6 (2001), pp. 1288–1290. ISSN: 02780046. DOI: 10.1109/41.969414.

[43] N. Alechina et al. "Using theorem proving to verify properties of agent programs". In: *Specification and verification of multi-agent systems*. Springer, 2010, pp. 1–33.

[44] V Nair and GE Hinton. "Rectified linear units improve restricted boltzmann machines". In: *Proceedings of the 27th international* (2010).

[45] O. Bastani et al. "Measuring Neural Net Robustness with Constraints". In: *CoRR* abs/1605.07262 (2016).

[46] Luca Pulina and Armando Tacchella. "An Abstraction-Refinement Approach to Verification of Artificial Neural Networks". In: Springer, Berlin, Heidelberg, 2010, pp. 243–257. DOI: 10.1007/978-3-642-14295-6_24.

[47] Luca Pulina and Armando Tacchella. "Challenging SMT solvers to verify neural networks". In: *AI Communications* 25.2 (2012), pp. 117–135. ISSN: 0921-7126.

[48] R. Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[49] R. Sutton. "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding". In: *Advances in Neural Information Processing Systems 8*. MIT Press, 1996, pp. 1038–1044.

[50] Edwin D. Reilly, Anthony. Ralston, and David. Hemmendinger. *Encyclopedia of computer science*. Wiley, 2003, pp. 405–419. ISBN: 0470864125.

[51] Reuters. *Reuters-21578 Dataset*. 1987. URL: http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.html.

[52] Stuart J. (Stuart Jonathan) Russell, Peter. Norvig, and Ernest. Davis. *Artificial intelligence: a modern approach*. Prentice Hall, 2010, p. 1132. ISBN: 9780136042594.

[53] S. Haykin. *Neural Networks and Learning Machines*. Pearson Education, 2011. ISBN: 9780133002553.

[54] S. Shapiro. *Specifying and verifying multiagent systems using the cognitive agents specification language (CASL)*. University of Toronto, 2005.

[55] K Scheibler et al. "Towards Verification of Artificial Neural Networks." In: *MBMV* (2015).

[56] Skymind. *A Beginner's Guide to Recurrent Networks and LSTMs*. 2016. URL: http://deeplearning4j.org/lstm.html (visited on 01/26/2017).

[57] Christian Szegedy et al. "Intriguing properties of neural networks". In: (Dec. 2013). arXiv: 1312.6199.

[58] V. Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *CoRR* abs/1312.5602 (2013).

[59] V. Nair and G. Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.

[60] W. Winston. *Operations research: applications and algorithms*. v. 1. Duxbury Press, 1987.

[61] Wayne L. Winston. *Operations research : applications and algorithms*. Duxbury Press, 1987, p. 1025. ISBN: 0871500655.

[62] X. Huang et al. “Safety Verification of Deep Neural Networks”. In: *CoRR* abs/1610.06940 (2016).

[63] Y. LeCun and C. Cortes. “MNIST handwritten digit database”. In: (2010). URL: `http://yann.lecun.com/exdb/mnist/`.

[64] Z. Kurd and T. Kelly. “Establishing safety criteria for artificial neural networks”. In: *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*. Springer. 2003, pp. 163–169.